

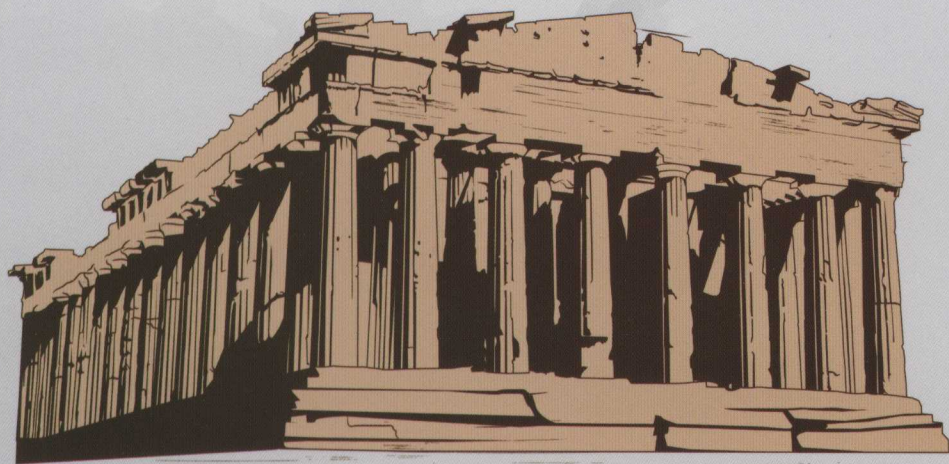
版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

互联网创业核心技术

构建可伸缩的Web应用

【美】Artur Ejsmont 著 李智慧 何坤 译



Web Scalability for Startup Engineers
Tips & Techniques for Scaling Your Web Application



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

Web Scalability for Startup Engineers

Tips & Techniques for Scaling Your Web Application

互联网创业核心技术

构建可伸缩的Web应用

【美】Artur Ejsmont 著 李智慧 何坤 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

可伸缩架构技术是所有互联网技术中最重要,也是最引人入胜的技术。本书针对互联网创业需求快速迭代,业务快速发展,短时间内用户、数据、访问量激增的特点,提纲挈领地描述了伸缩性架构的基本原理与设计原则,详细阐述了Web应用前端层、服务层、数据层的可伸缩架构,并花大量篇幅讲述了缓存技术和异步处理技术的可伸缩设计及其在Web系统中的具体应用。

本书面向互联网创业公司工程师,也适用于所有互联网行业的工程师,对非互联网行业的软件工程师也有借鉴作用。事实上,本书适合所有对可伸缩架构有兴趣的软件技术人员阅读。

Artur Ejsmont, Web Scalability for Startup Engineers tips&Techniques for Scaling Your Web Application, ISBN 9780071843652, Copyright © 2015 by McGraw-Hill Education.

All Rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including without limitation photocopying, recording, taping, or any database, information or retrieval system, without the prior written permission of the publisher.

This authorized Chinese translation edition is jointly published by McGraw-Hill Education and Publishing House of Electronics Industry. This edition is authorized for sale in China Mainland.

Copyright © 2016 by McGraw-Hill Education and Publishing House of Electronics Industry.

版权所有。未经出版人事先书面许可,对本出版物的任何部分不得以任何方式或途径复制或传播,包括但不限于复印、录制、录音,或通过任何数据库、信息或可检索的系统。

本授权中文简体字翻译版由麦格劳-希尔(亚洲)教育出版公司和电子工业出版社合作出版。此版本经授权仅限在中国大陆销售。

版权©2016由麦格劳-希尔(亚洲)教育出版公司与电子工业出版社所有。

本书封面贴有McGraw-Hill Education公司防伪标签,无标签者不得销售。

版权贸易合同登记号 图字:01-2015-7884

图书在版编目(CIP)数据

互联网创业核心技术:构建可伸缩的web应用/(美)阿特·艾斯蒙特(Artur Ejsmont)著;李智慧,何坤译. —北京:电子工业出版社,2016.12

书名原文:Web Scalability for Startup Engineers

ISBN 978-7-121-30112-4

I. ①互… II. ①阿… ②李… ③何… III. ①网页制作工具—程序设计 IV. ①TP393.092.2

中国版本图书馆CIP数据核字(2016)第247615号

策划编辑:刘 皎

责任编辑:郑柳洁

印 刷:北京季蜂印刷有限公司

装 订:北京季蜂印刷有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编:100036

开 本:720×1000 1/16 印张:21 字数:370.3千字

版 次:2016年12月第1版

印 次:2016年12月第1次印刷

定 价:89.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819 faq@phei.com.cn。



推荐序

这是一个最好的时代，我们赶上了互联网的大潮；这也是一个最坏的时代，我们习惯的旧秩序被互联网打破了；这是一个创新的时代，任何奇思妙想都有被广大人民接受成为现实的可能；这还是一个变革的时代，在“互联网+”的春风中，线下的思维转换为线上的思维。

这一切的推手，就是互联网。互联网催生了一种新型的生活方式，也催生了适应于互联网的一种新的人群。连接人群和线上生活的纽带就是一个一个的互联网及移动互联网的应用。作为应用的典型 IT 形态，互联网应用经历了独占型应用，SOA 应用，在云时代就是云原生（Cloud Native）的应用。构建一个好的可伸缩的应用，不仅仅是一个优秀工程师的职责，同时也代表了对一种生活方式的认可。在这种应用里，我们看到了电商应用的身影：任何一个电商购物狂欢节，背后是工程师制作的良好弹性应用面对使用量波峰波谷的挑战。

除了电商应用，任何一个 To C 的 APP 都会面临同样的挑战，要承载捉摸不定的业务量及快速推进版本的演进。作为引来的昂贵的流量，需要良好的后台承接并及时处理，推动良好的客户体验及带来真实的业务发展，从而帮助 C 端的创业者快速验证自己的商业模式并快速迭代。

本书就是这样一本很好的最佳实践。本书作者将自己的成功经验总结归纳出来，对弹性架构的概念，软件设计的原则，以及如何构建一个优质的互联网应用，进行了深入讲解。作为一位互联网电商的资深从业者，书中很多概念与电商行业的最佳实践不谋而合，有些

观点对电商网站的构建是非常好的建设性意见。

本书的译者李智慧作为互联网老兵，对互联网应用网站的架构设计有丰富的经验；同时作为大数据行业的资深从业者，对数据运营的观念理解得精准到位，结合互联网网站设计及数据运营的思路，推动了本书翻译的成稿，不仅对原著内容的理解到位，同时对原著细节的拔高有神来之笔。

本书适合互联网技术从业人员阅读，是电商行业、移动互联网行业中不可多得的好书。

杨海明

京东云平台事业部总经理



译者序

每个互联网创业者的心中都有一个独角兽或者纳斯达克的梦想,不过不论梦想有多么远大,都需要从第一行代码开始,一步一步地构建系统。那么,第一行代码和最后服务数亿用户的系统之间是什么关系呢?一个最初由两三个工程师开发的产品雏形,如何经过逐渐地重构、演化、迭代、伸缩,最终成为一个巨无霸的系统?

本书面向互联网创业公司工程师,讲述构建可伸缩 Web 系统的核心概念、关键原则和主要方法。本书第 1~5 章和第 9 章由李智慧翻译,第 6~8 章由何坤翻译。第 1 章和第 2 章讲述可伸缩系统的核心概念与软件基本设计原则。我强烈建议你认真阅读这两章,这部分内容包含了开发一个可伸缩的 Web 系统甚至开发一个良好软件的基本原理和设计原则,是其他一切技巧和方法的元规则。第 3~8 章讲述了构建可伸缩 Web 应用的主要方法和工具,这几章的内容相对比较独立,你可以挑选自己感兴趣的章节阅读,也可以在工作中遇到问题时快速浏览,寻找方法和灵感。第 9 章讲述可伸缩的系统运维及可伸缩的个人和团队,如果你正处在一个高速发展的创业团队中,如果你对从技术走向管理感兴趣,我相信你可以从本章的内容中收获多多。

诚如书中一再强调,可伸缩系统的设计是一种权衡的艺术,必须对每一种方案的优缺点都了如指掌,才能在面对实际问题时做出最合适的选择。高并发可伸缩系统的设计看似纷繁复杂庞大无比,实际上关键的核心技术也就那么几样,如果深入掌握了这些关键技术,就抓住了可伸缩系统设计的核心。这几样关键技术,可能需要在不同场景,从不同视角反复思考琢磨,才能真正掌握。我曾经在《大型网站技术架构:核心原理与案例分析》一书

中表达过这个观点，如果读过《大型网站技术架构：核心原理与案例分析》，那么推荐你阅读本书，从创业公司工程师的视角再次审视分布式网站设计的方方面面，掌握可伸缩 Web 应用设计权衡的艺术。

虽然罗马不是一天建成的，但是一名优秀的工程师，在放下建设罗马的第一块砖的时候，心中就已经有了整个罗马建成后辉煌的场景，以及整个建造历程。他运筹帷幄、胸有成竹，每一次的快速发展他都已经提前做好准备，每一次的剧烈变化他都有应对的方案。因为对艰难困苦了然于心，所以倍加从容。

本书就是这样一本书，它描述了一个愿景，当你的公司成为独角兽公司的时候，你的系统会长什么样子；它展现了一个历程，你该如何一步一步地将你的系统伸缩成一个巨无霸系统；它提供了一组工具和方法，你如何利用这些工具方法改造你的系统实现你的梦想。

星空就在你的头顶，路就在你的脚下，梦想就在你的前方，带上这本书，我们出发吧！

李智慧



关于作译者

关于作者

Artur Ejsmont 是一位充满激情的软件工程师及敏捷领导者，目前就职于雅虎悉尼。Artur 从事 Web 应用方面的开发工作已经超过十年，同时也大量关注敏捷及创业公司管理，热爱精益创业模型。Artur 开发的网站每小时 PV 数超过一百万，为欧洲及澳洲两个大洲的用户提供服务。他还在大学开设课程，分享他的技术与知识。作为一名科技作家、博客写手、技术演讲者，Artur 对技术社区有许多积极的贡献。

关于技术编辑

Bill Wilder 是 Finomial 公司的 CTO，偶尔在技术会议上演讲，也是一位不连续更新的博客写手，波士顿 Azure 的领导者（从 2009 年开始）。他被微软授予 Azure MVP（最有价值专家）殊荣，著作有《云架构模式》（2012 O'Reilly Media 出版）。你可以通过 Twitter 账号@codingoutloud 关注他。

TJ Wilder 是马萨诸塞艾姆赫斯特学院的学生，主修计算机科学，辅修数学。他从 12 岁开始编程，偶尔会被其他事情打断，比如吃饭、睡觉、唱歌。当 DraftKings.com 还是创业公司的时候，他就在那里实习。他期待本书能给那些在创业公司工作的人们更多帮助。

Dr. Danny Coward 是 Oracle 的首席架构师及 Web 架构师。Coward 是 Java EE、Java SE/JavaFX Web Sockets 的 Java API 的独立专家领导，也是 Oracle WebSocket 技术文档的独立作者。Coward 在 Oracle 的 WebSocket 方面的领导工作使他成为 Java WebSocket 编程方面的领袖级专家。Coward 是一个有着十多年经验的 Java 开发者，擅长将复杂的技术简化以实现业务对象。Coward 在 Java 软件的所有方面（从 Java Me 到 JavaFx 技术）都有着专家级的经验。

关于译者

李智慧，曾供职阿里巴巴及英特尔亚太研发中心，从事分布式系统与大数据方面的开发。《大型网站技术架构：核心原理与案例分析》作者。目前正在互联网方面创业。

何坤，宅米首席架构师，前阿里巴巴平台架构师。曾参与阿里中文站架构建设，以及 WebX 框架、云计算存储平台 Doris 等核心系统研发。曾就职丰联金融证券期货部任首席架构师。对大型互联网站架构有深刻理解，对中间件、分布式系统及新技术研发始终保持热情。目前关注自然语言处理、机器理解等新课题。

致谢

感谢 Bill Wilder、TJ Wilder、Danny Coward，以及 Brandi Shailer 在我写作本书期间给予的反馈和帮助。也感谢 Dion Beetson、Craig Penfold、Jackie Reses、Venkatesh Kanchan、Marcelo Maidana、Kristian Kauper、Christof Mueller、Ścibór Sobieski、Mariusz Jarocki、Andrzej Nowakowski，我的朋友们和家人们在过去 35 年间给予的支持和鼓励。没有他们就不会有我现在的成就。

本书献给所有热爱编程的你们，

Geek 们，

人类的未来真的就在你们手中，

就在此刻。



引言

近些年来，伴随着技术的进步，越来越多的 Web 应用系统需要存储、转化、处理越来越多的数据，而这必将要求工程师们掌握构建可伸缩的 Web 系统的能力。

当我了解到大多数工程师都缺乏这种构建可伸缩 Web 系统的能力时，我觉得有必要写一本与此有关的书。一方面，目前市面上缺乏相关的著作；另一方面，那些在小公司工作的工程师们也缺乏必要的环境去学习可伸缩架构的设计方法。因此，本书致力于讲解软件架构与基础设施如何协同工作，并最终实现系统的可伸缩性。

希望本书可以成为开启读者可伸缩架构设计之旅的一个里程碑。书中既给出了可伸缩架构设计的整体概览视图，同时也深入探讨了一些重要的技术点并给出了一些最佳实践建议。然而，书中并没有对每一个技术细节都深入阐述，而是尽可能地为读者呈现那些必要的概念、基本的规则，以及有意义的实例。

第 1 章：核心概念

作为本书第 1 章，本章主要介绍可伸缩性架构的一些核心概念及关于本书的一个概览性描述。本章通过描述可伸缩 Web 系统演化的不同阶段介绍相关的概念，同时，也会涉及一些基础设施与可伸缩 Web 系统架构方面的整体描述。

第 2 章：软件设计原则

本章主要探讨一些构建弹性可伸缩系统的设计原则。首先是那些最基础的概念，比如简化与解耦；其次是一些面向对象的设计原则，比如单一职责原则和依赖注入原则；最后

是那些与可伸缩性直接相关的设计概念，比如功能分割、数据分区，以及自我恢复。

第 3 章：构建前端层

本章内容主要集中在直接与客户端软件（比如 Web 浏览器及移动 APP）交互的那部分基础设施上。本章会深入解释前端管理状态的几种方式，以及构建可伸缩前端的几种重要组件，比如负载均衡、代理、内容分发网络（CDN）；之后会探讨自动化伸缩及几种部署场景。

第 4 章：Web 服务

本章主要探讨几种不同的 Web 服务架构的优缺点。解释 Web 服务设计原则并深入细节讲解基于 REST-ful API 的伸缩性技术。

第 5 章：数据存储层

本章会解释数据层伸缩性技术的若干核心要点。除了涉及 MySQL 这一类关系数据库相关的技术外，还会花大量篇幅探讨 Cassandra 这类 NoSQL 数据存储技术。本章会穿插讲解一些技术细节，比如数据分区（也常称作分片技术）、数据复制，以及最终一致性。比较几种不同的数据存储层拓扑结构及与此有关的技术挑战。

第 6 章：缓存

本章内容主要着眼于缓存，缓存是一种改善 Web 系统伸缩性和性能的关键性手段。主要讲解基于 HTTP 的几种不同的缓存技术，以及 HTTP 缓存的伸缩性技术。此外，还会阐述对象缓存技术及其常用的伸缩性技术。最后，花一些篇幅探讨缓存最佳实践，以帮助读者在使用缓存时做出更好的决策。

第 7 章：异步处理

本章会讲解有关消息与事件驱动架构方面的主题。首先，探讨异步处理的概念与优点及如何利用消息代理改善 Web 应用的可伸缩性；其次，重点论述异步系统的挑战与使用中可能出现的陷阱；最后，简要比较几种主流的消息平台以帮助读者在具体工作中做出最佳选择。

第 8 章：数据搜索

本章主要讨论数据搜索方面的有关问题。数据搜索与数据存储密切相关，随着数据集的快速增长，优化数据搜索与访问方面的技术变得越来越重要。首先，会讲解不同类型的索引是如何工作的；其次，会花一些篇幅描述数据建模如何有助于改善伸缩性，以及在诸如 Cassandra 之类的 NoSQL 数据存储系统中考虑数据建模。最后，做一些搜索引擎方面的介绍及讲述相关技术如何应用于 Web 应用系统。

第 9 章：伸缩性的其他维度

这是本书的最后一章，阐述一些扩展读者自身工作效率的有关概念，以方便读者更好地实现自我管理与团队成长。首先，强调自动化作为一项关键技术 in 提高工程效率方面的重要作用，探讨有关自动化测试、部署、监控与报警方面的话题。其次，分享一些项目管理方面的个人经验与观察，这些技能将帮助读者在创业中更好地生存下来。最后，对成长中的敏捷团队可能遇到的挑战做一些反思。

目标读者

本书的主要目标读者是软件工程师、技术经理、DevOps，以及系统工程师。对于在校学生而言也许会有一定难度，不过对于中等水平甚至是初级工程师而言，绝大多数内容都是比较容易理解的。

本书假设读者对于如何利用相关技术构建一个 Web 应用系统有一个基本的了解。不过阅读本书不需要拥有任何特定的编程技能，诸如 Java、PHP、JavaScript、C#或者 Ruby，因为伸缩性是 Web 应用开发中一个通用的挑战，与具体语言无关。另外，本书也假设读者了解 HTTP 协议是如何工作的，以及对 IP 网络、HTML、C/S（客户端/服务器）软件开发等有一个基本的概念。



目 录

1

核心概念

1

什么是伸缩性.....	2
从单一服务器到全球用户的 Web 架构演化.....	4
单一服务器.....	5
使用更强的服务器：垂直伸缩.....	6
服务分离.....	10
内容分发网络：静态内容的伸缩性.....	12
分散访问流量：水平伸缩.....	13
服务全球用户的伸缩性架构.....	16
数据中心基础设施架构概览.....	18
前端.....	19
Web 应用层.....	20
Web 服务层.....	20
附加组件.....	21
数据持久层.....	21
数据中心基础架构.....	22
应用架构概览.....	23
前端.....	24
Web 服务.....	25
支撑技术.....	29
小结.....	30

2

软件设计原则

31

简单.....	31
隐藏复杂与构建抽象.....	32
避免过度设计.....	33
尝试测试驱动开发.....	34
从软件设计的简化范例中学习.....	35
低耦合.....	36
促进低耦合.....	37
避免不必要的耦合.....	39
低耦合范式.....	40
不要重复自己 (DRY)	41
复制粘贴代码.....	42
基于约定编程.....	43
画架构图.....	46
用例图.....	49
类图.....	50
模块图.....	51
单一职责.....	52
改善单一职责.....	52
单一职责的例子.....	53
开闭原则.....	53
依赖注入.....	55
控制反转 (IOC)	57
为伸缩而设计.....	59
增加副本.....	60
功能分割.....	62
数据分片.....	63
自愈设计.....	65
小结.....	67

3

构建前端层

69

状态管理.....	70
管理 HTTP 会话.....	73
管理文件.....	77
管理其他类型的状态.....	80
可伸缩的前端组件.....	83
DNS.....	84
负载均衡器.....	85
Web 服务器.....	92
缓存.....	93
自动伸缩.....	94
部署案例.....	96
AWS 场景.....	97
私有数据中心.....	98
小结.....	101

4

Web 服务

102

Web 服务设计.....	102
Web 服务作为一种备用表示层.....	103
API 优先方式.....	105
务实的方式.....	107
Web 服务类型.....	108
以功能为中心的服务.....	109
以资源为中心的服务.....	111
伸缩 REST Web 服务.....	115
保持服务无状态.....	115
缓存服务响应.....	121
功能分割.....	124
小结.....	127

5 数据存储层 129

MySQL 伸缩性.....	130
复制.....	130
数据分区（分片）.....	142
NoSQL 伸缩性.....	157
最终一致性.....	160
快速恢复增加可用性.....	164
Cassandra 拓扑结构.....	166
小结.....	170

6 缓存 171

缓存命中率.....	171
基于 HTTP 的缓存.....	173
HTTP 缓存头.....	174
HTTP 缓存技术类型.....	179
伸缩 HTTP 缓存.....	185
缓存应用对象.....	188
对象缓存的一般类型.....	189
伸缩对象缓存.....	194
缓存的经验法则.....	198
缓存整个调用栈.....	198
用户间缓存重用.....	199
从哪儿开始使用缓存？.....	201
缓存失效的困难.....	201
小结.....	203

7 异步处理 204

核心概念.....	204
同步处理的例子.....	205
异步处理的例子.....	208
购物类比.....	211

消息队列.....	213
消息生产者.....	214
消息代理.....	215
消息消费者.....	216
消息协议.....	220
消息基础设施.....	221
消息队列的好处.....	224
实现异步处理.....	225
更好的伸缩性.....	226
平衡流量峰值.....	227
失败隔离和自我修复.....	228
解耦.....	229
消息队列相关的挑战.....	230
消息无序.....	230
消息重新入队列.....	233
竞态条件可能性增大.....	233
复杂度风险.....	234
消息队列有关的反模式.....	235
将消息队列当作 TCP 套接字.....	235
将消息队列当作数据库.....	235
耦合消息生产者和消费者.....	235
缺少坏消息处理.....	236
消息平台快速比较与选择.....	237
亚马逊简单队列服务.....	237
RabbitMQ.....	240
ActiveMQ.....	242
最后的比较说明.....	243
事件驱动架构介绍.....	245
请求/响应交互.....	246
直接队列交互.....	247
事件驱动交互.....	247
小结.....	250

8 数据搜索 252

索引介绍.....	252
数据建模.....	260
NoSQL 数据建模	260
宽列数据存储的例子	264
搜索引擎.....	271
搜索引擎介绍	272
使用专用搜索引擎	274
小结.....	275

9 伸缩性的其他维度 277

自动化实现生产力可伸缩.....	278
测试.....	278
构建与部署.....	280
监控与报警.....	285
日志聚合.....	289
个人可伸缩.....	291
加班不是一种伸缩性方案	291
自我管理.....	293
伸缩敏捷团队.....	300
增加人手.....	300
流程与创新.....	301
团结的文化.....	302
小结.....	303

A 推荐阅读 304

1

核心概念

创业公司需要面对非常多的不确定因素。如果想打造一家成功的创业公司，必须拥有一个极具弹性的系统，必须在快速变化的内外部环境保持快速响应的能力。这就要求创业公司的软件开发团队必须拥有构建可伸缩系统的能力。那些在传统企业中需要花费整年时间才能搞定的事情，创业公司可能必须要在几个星期内搞定。如果你足够成功并且足够幸运，你可能需要在几个星期内把你的系统的处理能力扩大十倍。当然，也可能在几个月后又把系统的处理能力缩小回去。

伸缩性设计对于任何工程师而言都是一件棘手的事情，而在创业公司中，伸缩性设计又面临着许多特别的挑战。就伸缩性技术本身而言，目前有很多公司提供这方面的基础服务，比如亚马逊云、微软云、Google 云，以及阿里云、腾讯云、UCloud 等，这些云服务可以让你完全不必考虑伸缩性方面的问题而只需关注自身的业务需求。当我们在本书中讨论伸缩性概念的时候，我们也会关注这些云服务是否适合创业公司所面临的挑战。要完全理解伸缩性需要一个渐进的过程，为了简化问题，我会在开始的时候在一个比较高的层面上讨论一些核心的概念。任何人在对 Web 应用开发有一些基本了解的基础上就可以轻松地阅读本书。在后面的章节中，我会深入阐述每个概念的详细细节。现在，我们首先要对伸缩性有一个基本的认知：什么是伸缩性，以及系统伸缩性如何演化；什么是大型应用；什么是大型应用架构。

要想完全掌握本章中提及的概念，可能需要读者在完整阅读本书以后，再回过头重新回顾本章内容。开始的时候，有些概念可能会非常抽象难以理解，但是当你对伸缩性架构有了一个系统性的认识以后，就会发现这些概念可以被完美地组合在一起，形成一个有机的整体。本章中也包含了一定数量的图例，如果你只是匆匆一扫而过，可能会忽略很多有用的信息，最好的办法是在阅读过程中把这些基础设施图及架构图再画一次，这不仅有助于理解本书的内容，也有助于你在下一次找工作面试时表现得更出色。

什么是伸缩性

在深入讨论核心概念之前，让我们先对伸缩性有一个统一的定义。你愿意阅读本书就是因为你想让你的 Web 应用系统变得可伸缩，或者变得更可伸缩。但是可伸缩究竟是什么意思呢？

伸缩性是指系统可以根据需求和成本调整自身处理能力的一种能力。伸缩性常常意味着系统可以改变自身的处理能力以满足更多用户访问、处理更多数据而不会对用户体验造成任何影响。此外，还有一件重要的事情必须提醒工程师们注意，伸缩性不仅需要伸（增强系统处理能力，即扩容），有时候也需要缩（缩减系统处理能力，即缩容）。同时，这种伸缩还必须相对比较省钱和快速。

正如在不同场景下对伸缩性有不同要求，可以在不同维度上对伸缩性进行度量。伸缩性主要从以下几个方面度量。

处理更多数据

这是最常见的一种挑战。如果你的业务蒸蒸日上，你的应用越来越受欢迎，那么你就需要处理越来越多的数据。你将不得不处理更多的用户账号、更多的产品、更多的地理信息数据，以及更多的数字化内容。这些数据需要存储、需要检索、需要写入磁盘又从磁盘读出来、需要通过网络传输，因此处理这些数据会给你的系统带来更大的压力。特别是随着大数据分析变得越来越流行，企业对数据变得越来越“贪心”，以前被随意丢弃的数据现在都要被存储和处理。

处理更高的并发

度量并发的指标通常是：应用系统可以同时服务多少个用户。对于一个 Web 应用系统而言，并发度的意思是最多有多少个用户可以同时访问你的网站而不会感到访问速度变慢。由于服务器的 CPU 数目是有限的，能同时运行的线程数也是有限的，因此处理高并发是一件非常有挑战的事。如果你还要同步某些代码的执行以保证数据的一致性，那么这个挑战将变得更加艰难。更高的并发也意味着系统需要同时打开更多的连接，启动更多线程，处理更多消息，CPU 需要更多次上下文切换。

处理更高频次的用户交互

伸缩性的第三个维度是你的系统和你的用户之间的交互频次。这个维度和并发度看起来很像，但是稍有不同。交互频次衡量你的用户和你的服务器交换信息的频繁程度，例如如果你做的是个 Web 站点，用户大概需要每 15 秒或 2 分钟不等从一个页面跳转到另一个页面；但是如果你做的是个多人交互的移动游戏，用户可能需要每秒钟和服务器通信好几次。因此，交互频次相对独立于用户并发度，而且更依赖于应用的类型。与交互频次相关的最大挑战是响应得延迟。如果应用的交互频次在增长，你就必须让应用响应得更快，这就要求系统有更快的读写速率进而将系统并发度推到一个更高的高度。

通常会综合上述三个维度去考量一个系统的伸缩性。一般说来，实现系统伸缩性的过程中，扩容比缩容更重要也更常见。虽然我们都期望尽量减少浪费并降低不必要的无效工作，但是对于创业公司，由于商业需求的变更而导致的无意义的系统伸缩和浪费在所难免。

可能你已经注意到，伸缩性和性能密切相关，但它们并不是一回事。性能更多的是衡量系统处理一个请求或者执行一个任务需要花费多长时间，而伸缩性则更关注系统是否能够随着请求数量的增加（或减少）而相应地拥有相适应的处理能力。

举个例子，你有 100 个并发用户，每个用户平均每 5 秒发送一个请求，那么系统吞吐量就是每秒 20 个请求。性能指的就是系统需要花费多少时间处理这每秒 20 个请求；伸缩性指的是在不影响用户体验的前提下，系统最多能够处理多少个并发用户及用户最多能发送多少个请求。

最后要说的是，软件产品的伸缩性也许会受限于软件开发团队的规模。如果要系统具

有伸缩性,那么对应的软件开发团队也必须具有伸缩性,否则就没有足够的工程师资源去快速响应需求的变更。尽管看起来软件开发团队的伸缩性和技术无关,但事实上系统架构和设计会影响团队规模。如果你的系统设计是紧耦合的,所有人都在同一份代码上工作,那么你就很难扩展你的工程师团队。由于团队的沟通效率和团队规模成指数关系,因此当完成同一个工作的技术团队的规模达到 8~15 人时,效率就会变得非常低。^[40]

提示

为了更好地体会伸缩性对创业公司的影响,让我们试着从公司视角观察。问你自己:“制约我们公司持续发展的的问题有哪些?”。答案不仅包括我们前面提到的处理高吞吐量导致的技术架构方面的问题,也包括开发过程、团队、代码结构等一系列问题。本书第 9 章将详细解释这些非技术架构方面的伸缩性问题。

从单一服务器到全球用户的 Web 架构演化

就像大多数创业公司刚起步时一样,我刚入行时经常会开发一些只有一台服务器的 Web 应用。在我的职业生涯中,我经历过许多不同种类的公司,因此也见证了各种 Web 应用的伸缩性演化阶段。在我们深入讨论伸缩性之前,我想展示这些不同阶段的演化过程以更好地解释:一台放在你桌子底下的服务器如何随着业务的发展演化成遍及全球的成千上万台服务器。

本章将在一个较高的层面讨论问题,而在后续的章节中逐步深入到细节。讨论演化阶段也能让我们有机会接触更多的概念并逐步深入到一些更复杂的话题。需要注意的是,这里展示的很多伸缩性演化架构阶段仅仅在你开始做规划时有参考意义。很多情况下,真实世界的系统并不完全按照这个方式去演化,更可能会经历多次重写。还有一些时候,系统在设计 and 诞生之初就处于某个演化的特定阶段并一直保持不变,或者在发现架构制约的时候直接向上跳跃一到两个阶段而不是逐步演化。

提示

尽量避免完全重写应用,特别是创业公司。^[45] 重写总是比你预期的时间更长,也比你预估的难度更大。基于我的经验,一次重写带来的麻烦需要两年才能终结。

单一服务器

让我们从部署单一服务器开始,这可能是最简单的一种配置也是很多小项目最开始的方式。在这种场景中,整个应用运行在一台单一的服务器上。如图 1-1 所示,所有的用户请求都被一台服务器处理。一般说来,DNS(域名服务系统)作为一种付费服务由主机服务商提供,DNS 也不运行在你的个人主机上。这种场景下,用户通过连接 DNS 获得你部署的 Web 应用的服务器主机的 IP 地址。一旦获得 IP 地址,用户就可以直接发送 HTTP(超文本传输协议)请求到你的 Web 服务器。

由于只部署了一台服务器,整个应用的所有计算都只能由一台服务器处理。这台服务器上面也许部署了一个数据库管理系统(MySQL 或者 Postgres),同时还运行着图片服务。当然,Web 应用程序也在上面运行。

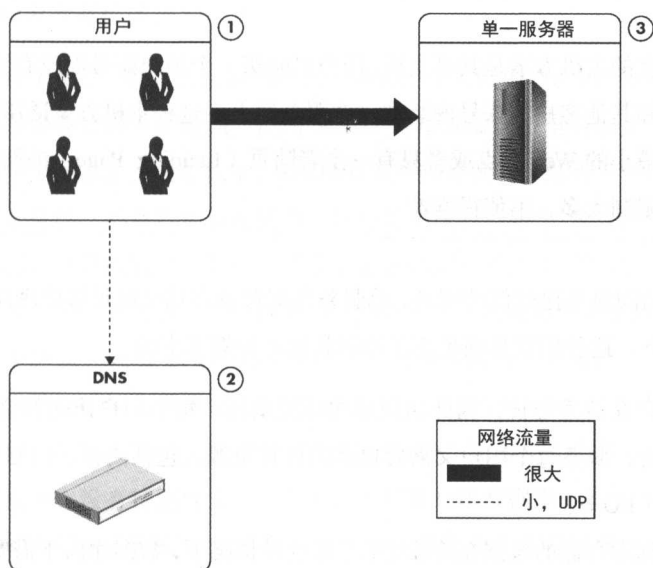


图 1-1 单一服务器

图 1-1 所示为单一服务器情况下的网络流量分布。用户端首先连接 DNS 服务器对网站域名进行解析,获得网站的 IP 地址,然后开始对网站服务器发起一系列的请求。网站服务器需要响应各种 Web 页面、图片、CSS 文件和视频文件,所有这些响应都只在这一台服务器上处理,所有网络流量也都只经过这一台服务器进行传输。图中用不同粗细的线

条箭头标示网络流量的大小。

这种应用是一个典型的小公司的 Web 网站，网站部署了一个产品目录系统、一个博客系统、一个论坛或者一个内部服务 Web 应用。这种网站甚至不需要用一个专门的服务器，只需要在一个共享主机上用 VPS（虚拟专用服务器）就够了。

虚拟专用服务器（VPS）是主机供应商们用的一个术语，指那些用以出租的虚拟机。当你购买一个 VPS 实例时，你的 VPS 实例和其他 VPS 实例将共用一个物理主机。VPS 用起来就像一个正规服务器——你拥有自己的操作系统和全部的权限。因为多个 VPS 实例可以同时运行在同一个物理服务器上，所以 VPS 比专用服务器便宜。同时，由于 VPS 比较便宜，又容易快速升级（你只需要按一下按钮就可以增加更多内存和 CPU），所以是创业公司非常好的选择。

最便宜的主机方案是共享主机，用户只购买一个用户账号而没有管理权限，这个账号和其他客户的账号安装在一个服务器上。这种主机方案适用于那些只需要一个最小的 Web 站点或者只有一个着陆页（Landing Page）的网站，但是这种方案限制太多，不值得推荐。

对于一个访问流量比较低的网站，单服务器配置也许就已经足够处理用户请求了。然而，多数情况下，这种配置是满足不了你的长远可伸缩需求的。

- 你的用户在持续增长，因此访问量在持续增长。每个用户访问都会对服务器造成负载压力，服务每个用户又需要更多的计算资源，包括内存、CPU 时间，以及磁盘读写（I/O）。
- 你的数据库存储的数据在持续增长。在这种情况下，数据库由于需要更多的 CPU、内存和 I/O 请求而变慢。
- 你要扩展系统增加新的功能，这使得每一次用户请求都要消耗更多的系统资源。
- 上面这些因素常常会叠加在一起。

使用更强的服务器：垂直伸缩

一旦你的应用达到服务器的极限（由于网络流量、数据处理规模或者并发度等因素的

增长), 就必须决定如何去伸缩系统。有两种不同的伸缩性方案: 垂直伸缩和水平伸缩。这两种伸缩性技术本书都将讨论, 不过由于垂直伸缩概念更简单, 在演化的第一阶段使用得更普遍, 所以我们优先讨论垂直伸缩。

通过升级硬件和网络吞吐能力可以实现垂直伸缩。由于不需要改变应用架构, 所以通常被认为是最简单的短期伸缩性方案。如果你的服务器是 8GB 内存, 仅仅通过替换硬件就可以轻松升级到 32GB 甚至 128GB。你不需要改变应用系统的运作方式或者添加一些抽象层去实现系统可伸缩。如果你的应用部署在虚拟服务器上, 那么垂直伸缩也许只需要点几下鼠标, 下载一个升级虚拟服务器实例的订单即可。

垂直伸缩的方案有很多:

- 通过使用 RAID (独立冗余磁盘阵列) 增加 I/O 吞吐能力。I/O 吞吐量和磁盘存储是数据库服务器的主要瓶颈。使用 RAID 并增加磁盘数量有助于将数据读写请求分布到多个磁盘上。最近几年, RAID10 变得格外流行, 这种 RAID 方案既提供了数据冗余存储又提高数据吞吐能力。从应用角度看, 整个 RAID 看起来就是一个数据卷标, 但是在底层实际包含了多个磁盘共同对外提供读写访问。
- 通过切换到 SSD (固态硬盘) 改善 I/O 访问速度。随着固态硬盘技术越来越成熟, 价格越来越低, SSD 也变得越来越流行。根据不同的基准测试方法, SSD 随机读写速度大约比传统磁盘快 10 到 100 倍。应用通过替换 SSD 硬盘可以节约更多 I/O 等待时间。不过不幸的是, 顺序读写速度提升就没有这么高了, 所以现实中应用性能提升也没有特别巨大。事实上, 大多数开源数据库 (比如 MySQL) 会优化数据结构和算法, 尽可能多地执行顺序硬盘操作而不是随机操作。而某些数据存储系统, 比如 Cassandra, 做得更甚, 所有的写操作和多数的读操作都只使用顺序 I/O 操作, 这也使得 SSD 更缺乏吸引力。
- 通过增加内存减少 I/O 操作 (如果你的应用部署在独立的物理服务器上, 128GB 内存是一个比较实惠的常规配置)。增加内存意味着文件系统有更多的缓存空间, 应用程序有更多的工作内存。此外, 内存大小对于数据库服务器效率的提升也格外重要。

- 通过升级网络接口或者增加网络接口提高网络吞吐能力。如果你的服务器需要处理大量视频等媒体类内容，也许需要升级网络供应商连接，甚至升级网络适配器以获得更高的吞吐能力。
- 更新服务器获得更多处理器或者更多虚拟核。拥有 12 或者 24 线程（虚拟核）的服务器是一个比较实惠且合理的升级方案。CPU 和虚拟核越多，能够同时执行的进程数就越多，系统因此变得更快，这不仅仅是因为多个进程可以不必共享同一个 CPU，还因为操作系统不需要在同一个核上执行多个进程进而执行不必要的上下文切换。

垂直伸缩是一个很好的选择，特别是应用很小或者你能负担得起硬件升级的情况下。垂直伸缩实现简单是主要的优点，使用这种方案，你不需要重构任何东西。不幸的是，垂直伸缩也伴随着一些严重的制约，一个主要的制约是成本，当越过某个点后，垂直伸缩会变得格外昂贵。^[43]

图 1-2 所示为每计算单元价格和总计算需求之间一个近似的关系。可以看到，起初，扩容伸缩还是比较便宜的，但是当越过某个点后，增加计算能力就变得格外昂贵。举个例子，128GB 内存需要 3000 元，256GB 内存就需要 18000 元，价格增长远不止翻一番。

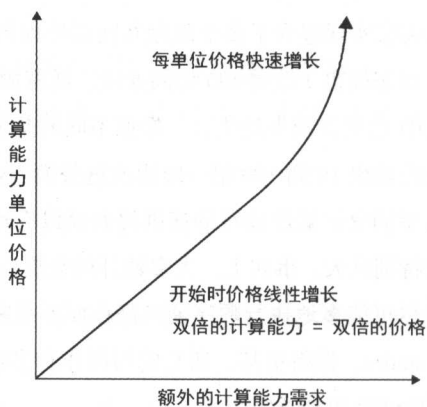


图 1-2 伸缩性单位成本

第二个比较大的问题是垂直伸缩是有极限的。无论你愿意花多少钱，内存都不可能无限地增加下去。类似的限制还有 CPU 的速度，每台服务器的虚拟核数目，硬盘的速度。简单说来，到了某个极限，没有任何硬件能力能够继续增加。

最后，操作系统的设计或者应用程序自身制约着垂直伸缩最多只能达到某个点。举个例子，你不能通过增加 CPU 数目而无限增加 MySQL 的处理能力，因为同时锁竞争也会增加（特别是如果你使用了 MyISAM 这种比较老的 MySQL 存储引擎的话）。

如果多个线程共享诸如内存、文件这类资源时，需要用锁进行同步访问。低效的锁管理会导致锁竞争成为瓶颈。应用操作应该使用细粒度的锁，否则，应用需要花费很长的时间去等待释放锁。一旦锁竞争成为瓶颈，增加再多的 CPU 核数也不会改善应用的处理能力。

高性能的开源应用或商业应用可以扩容到几十个 CPU 核，然而，在购买新硬件前最好还是确认下系统的扩容伸缩能力。由于高效的锁管理是一项有挑战的任务，需要大量的经验和详细的调试，所以自己开发的应用通常在锁竞争方面做的差一点。在一些极端的情况下，因为在设计之初就没有考虑任何高并发的场景，结果导致增加 CPU 核对应用处理能力提升没有任何帮助。

如图 1-3 所示，垂直伸缩不会对系统架构产生任何影响。你可以垂直伸缩扩容任何一台服务器、网络连接或者路由器而无须修改任何代码或者重构任何东西。唯一要做的就是用更强大更快速的硬件替换现有的硬件。

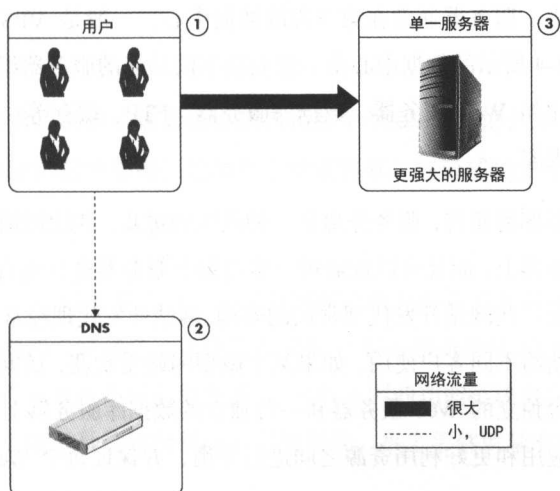


图 1-3 单一服务器，但是是更强大的服务器

服务分离

在网站发展早期,垂直伸缩不是唯一的解决方案。另一个简单的方案是通过在不同的物理机上安装不同类型的服务,使得一个系统的不同部分被分离部署在不同物理服务器上。在这种场景下,一个服务可以是一个类似 Web 服务器这样的应用(比如 Apache),或者是一个数据库引擎(比如 MySQL)。这就使得应用服务器和数据库分离到不同的服务器上。

类似地,你也可以把 FTP、DNS、缓存及其他服务部署在不同的物理机器上。对单一服务器部署进行可伸缩扩容,对可分离的服务进行分隔部署是一种比较轻的解决方案。不过,这种伸缩方案并不能一直持续进行下去,一旦你的所有服务类型都已经分别部署在独立的服务器上,你就没办法继续用这种方法扩容下去了。

缓存是一种重要的服务,目标是通过快速返回提前生成好的内容降低请求响应延迟。缓存是构建可伸缩系统的一种重要技术,将在第 6 章讨论更多关于缓存的细节。

图 1-4 所示为不同类型的服务分离部署在不同服务器上的一个高层次基础架构视图。虽然看起来跟单一服务器部署很像,但是这种架构方案可以不断增加服务器数量进而分摊整个系统的负载压力。服务器通常在第三方的数据中心,一般是 VPS、租借的硬件、专有的服务器等。图 1-4 所示的数据中心由一组安装不同功能的服务器组成。每个服务器都承担不同的角色,比如 Web 服务器、数据库服务器、FTP、缓存等。本章将讨论数据中心布局的一些详细内容。

对于单一服务器部署而言,服务分离是一种巨大的进步。相比以前,你可以将负载压力分摊到更多的服务器上,而且可以按需进一步对每个服务器进行垂直伸缩。这种架构方案是很多小型网站及一些网站开发代理常用的架构。网站开发代理经常在一组共享服务器上部署很多微型网站给不同客户使用。如果某个微型网站受欢迎,访问量大,就会把它分离出来,部署在一台独立的 Web 服务器和一台独立的数据库服务器上。开发代理机构可以因此在满足客户应用和更好利用资源之间进行平衡,并保证每个 Web 应用都比较简单而整体又比较公平。

类似于开发代理机构在不同服务器上部署不同客户应用,你也可以把这种技巧应用到自己的网站中,将 Web 应用分成多个较小的独立的部分并把它们部署在不同的服务器上。例如,如果你的应用有一个管理控制台模块,用户可以在这里管理他们的账号,那么你就可以把它分离出来形成一个独立的 Web 应用,然后把它部署在一个独立的服务器上。

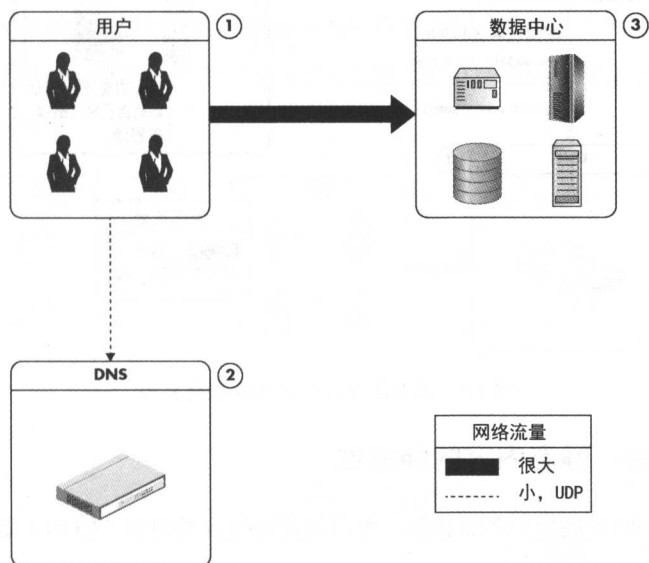


图 1-4 不同的服务部署在不同的服务器上

提示

服务分离背后的核心理念是你应该将整个 Web 应用切分成一组不同的功能模块,然后将它们独立部署。这种基于功能将系统划分成独立可伸缩的模块的方式被称为功能分割。

图 1-5 所示为一个 Web 应用利用功能分割将负载分布在更多服务器的场景。应用的每个部分都使用不同的二级域名,这样就可以基于 Web 服务器的 IP 地址进行流量分发。不同的功能模块也许部署在不同的服务器上,这些不同的功能模块也会有不同的垂直伸缩需求。显然,一个系统越是能够分割成不同的部件,每个部件就越有弹性越好。

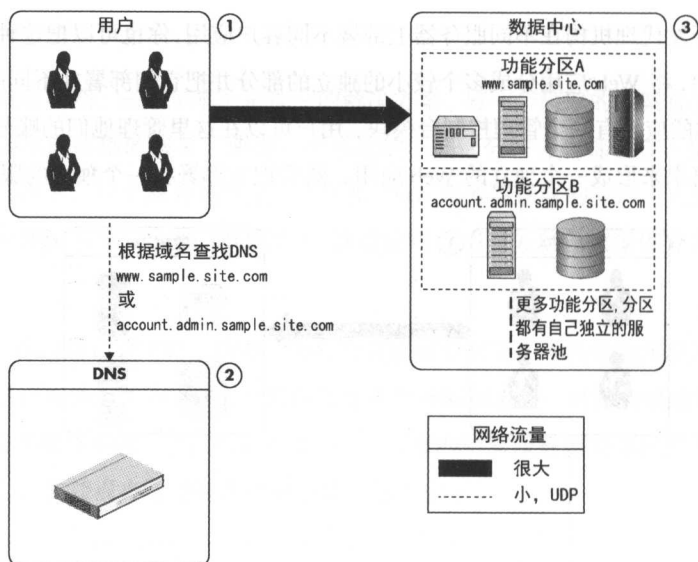


图 1-5 应用按照功能进行分割的架构

内容分发网络：静态内容的伸缩性

随着应用不断发展用户不断增长，可以通过内容分发网络（CDN）服务减轻应用网络流量负载压力。

内容分发网络是一种提供静态文件（比如图片、JavaScript、CSS 及视频）全球分布的服务。它的工作原理有点像 HTTP 代理。用户如果需要下载图片、JavaScript、CSS 或者视频内容，可以通过连接 CDN 服务器下载而不是连接应用服务器。如果 CDN 服务器没有用户需要的内容，CDN 服务器会请求应用服务器获取这部分内容然后再缓存到 CDN 服务器上。一旦文件缓存在 CDN 服务器上，后面的用户就再也不需要连接应用服务器了。

通过将应用集成到某个 CDN 服务，可以显著减少应用服务器需要的网络带宽。你将可以用更少的 Web 服务器提供 Web 应用静态内容。最后，由于 CDN 提供商通常是在全球范围内提供数据内容本地化的公司，因此你的用户也会获得更好的资源本地化服务。如果你的数据中心部署在北美，而你的用户从欧洲访问，不可避免会有很大的响应延迟。这

种情况下，CDN 将会从距离用户最近的服务器提供静态内容，进而加速这些用户的页面加载时间。

图 1-6 所示为一个集成了 CDN 服务的 Web 应用系统架构。首先，用户连接 DNS 服务器。然后，从你的数据中心服务器请求 Web 页面，CDN 服务器加载页面附加的资源，比如图片、CSS 和视频。最终，你的服务器和网络的访问流量将会大大缩减，而且 CDN 通过各种优化手段可以使得提供这些网络访问流量比你自己提供更便宜。第 6 章将对 CDN 做进一步的详细解释。

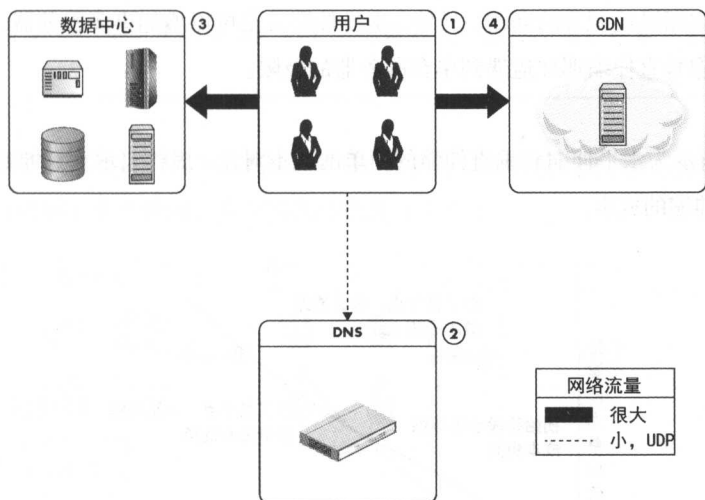


图 1-6 集成 CDN 的架构

这里有一件重要的事情需要指出，这是我第一次在伸缩性架构中提及第三方服务。我们不是一定要增加很多的服务器或者学习如何对 HTTP 代理进行伸缩。我们只需要简单地使用第三方的服务，然后依赖它提供的伸缩性能力就可以了。虽然这看起来有点像“伸缩性游戏的小把戏”，但它真的是非常强大的手段，特别是在创业公司的早期开发阶段，可能根本就没有足够的时间和金钱去调查这些技术。

分散访问流量：水平伸缩

目前为止，我们讨论的所有伸缩性架构演化都只是在单一服务器基础上做简单的修改。然而，水平伸缩则非常难以实现，很多情况下，必须要在应用开发之初就考虑周详。

虽然在某些情况下，水平伸缩可以通过修改应用架构在后期进行“添加”，但是大多数情况下，必须付出相当的开发代价。本书将讲述各种不同的水平伸缩性技术，但是现在，让我们考虑不同的组件运行在多台服务器上，而且无论是否有必要都还可以继续添加更多的服务器。一个真正意义上的可伸缩系统不需要很强大的服务器，甚至相反，它们运行在大量的廉价商业服务器上，而不是少量的强大的服务器上。

水平伸缩是指通过增加服务器提升计算能力的一类架构方法。水平伸缩被认为是伸缩性的圣杯，水平伸缩可以克服垂直伸缩带来的单位计算成本随计算能力增加而迅速飙升的问题。另外，水平伸缩总是可以增加更多服务器，这样，就不会像垂直伸缩那样遭遇到单台服务器的极限。

图 1-7 所示为水平伸缩和垂直伸缩的简单的成本对比。虚线表示垂直伸缩的成本，实线表示水平伸缩的成本。

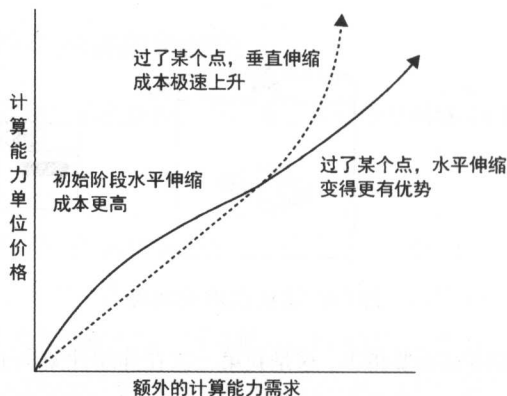


图 1-7 水平伸缩与垂直伸缩成本对比

水平伸缩技术带来的好处要在系统发展的后期才能体现出来。一开始，水平伸缩因为技术比较复杂需要更多的工作量，会花费更多的成本。一方面，这些成本体现在可水平伸缩的架构比基本的架构需要部署更多的服务器；另一方面，水平伸缩的架构需要更有经验的工程师去构建并维护它们。不过，一旦你的系统的计算能力达到某个点，水平伸缩就变成一个更好的策略。使用水平伸缩，可以不必花费购买顶级服务器的高昂的价格，也不会触及垂直伸缩会出现的天花板（买不到更强大的硬件）。

水平伸缩使用 CDN 这样的第三方服务不仅更节约成本，而且更透明。你需要的网络流量越多，需要付给 CDN 供应商的钱就越多，但是单位计算成本不变。这就意味着如果你的访问请求翻了一番，那么你支付的成本涨了一倍。更妙的是，对于某些服务供应商，随着你的资源需求增加，单位价格会下降。例如，亚马逊 CloudFront，前 10TB 传输数据每 GB 0.12 美元，超过这个数字后，每 GB 下降到 0.08 美元。

提示

云服务商愿意为高流量的客户提供更低的价格，因为这些客户之前的付费已经覆盖了必要的维护集成成本。对于访问量很大的客户，他们也确实很在意价格高低。

让我们快速浏览到目前为止的高层架构图。我们将一个系统的不同部分部署到不同服务器上，然后再加上水平伸缩，高层架构图如图 1-8 所示。

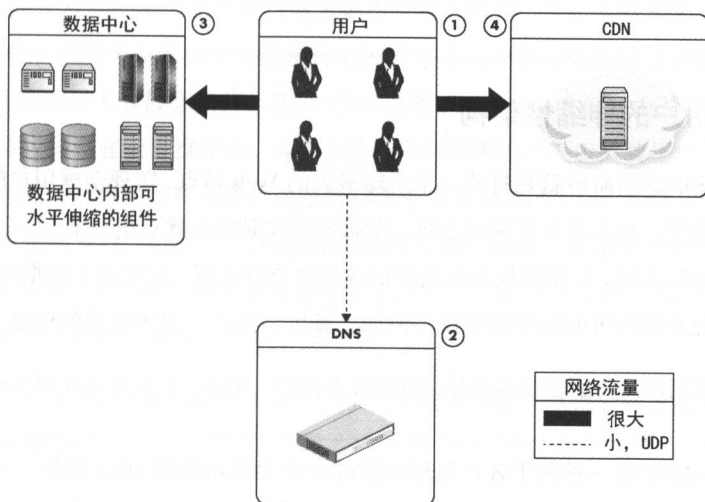


图 1-8 每种组件都有多台服务器

具有水平伸缩特性的系统和先前架构演化阶段提及的系统的不同之处在于：数据中心的每一种服务器角色都可以通过增加服务器进行扩容伸缩。事实上，在不同演化阶段可以部分地实现水平伸缩，有的服务实现水平伸缩，而有的服务则不实现。前面提到，达到真正意义上的水平伸缩很困难而且需要丰富的经验。因此，构建水平伸缩的系统先从那些容

易做到的地方做起，比如 Web 服务器、缓存；暂缓那些难以做到的地方，比如数据库及其他的持久存储。

在演化的这一阶段，一些应用使用轮询 DNS 服务实现 Web 服务器流量分发。轮询 DNS 不是实现 Web 服务器流量分发的唯一手段，我们将在第 3 章进一步讨论更多的方式。

轮询 DNS 是 DNS 服务器的特性之一，它允许将一个域名解析到多个 IP 地址中的一个。一般的 DNS 服务器会将一个域名（比如 ejsmont.org）解析到一个单一的 IP 地址（比如 173.236.152.169）。然而，轮询 DNS 允许将一个域名映射到多个 IP 地址上，每个 IP 地址都指向不同的机器。因此，每次用户请求域名解析，DNS 都会返回这些 IP 地址中的一个。目的就是将每一个用户的访问流量分发到 Web 服务器集群中的某一台上，不同的用户在不知情的情况下连接到不同服务器上。一旦用户接收到一个 IP 地址，他就会只与这台选中的服务器通信。

服务全球用户的伸缩性架构

架构演化的最后阶段就是打造一个全球最大的 Web 站点，实现全球用户的可伸缩性。一旦你服务的用户从几百万扩展到全球，你需要的数据中心就不止一个。一个数据中心可以部署很多的服务器，但是其他大洲用户的体验可能并不好，而且多个数据中心也能让你比较从容地应对那些可能的宕机事件（例如，暴风、洪水、火灾引起的停电）。

服务全球用户需要面临很多挑战，用到很多技巧，其中一个技巧是使用 GeoDNS 服务。

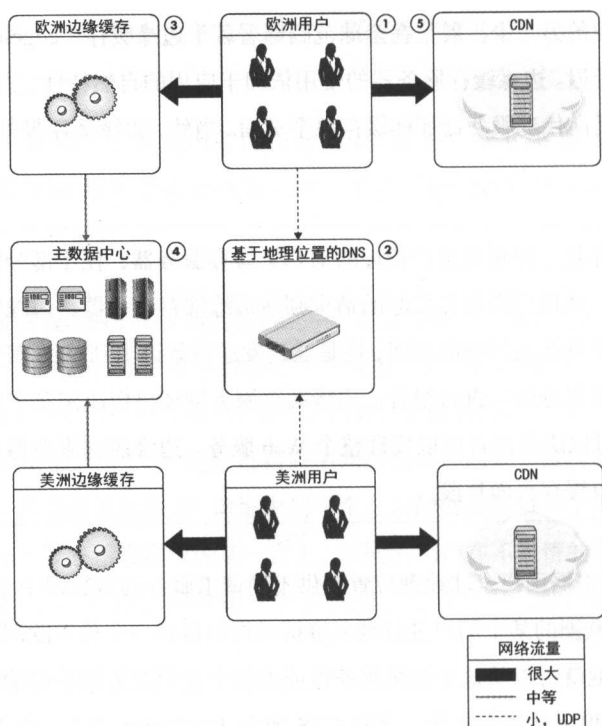
GeoDNS 是一种基于客户地理位置进行 IP 地址解析的 DNS 服务。一般 DNS 服务器收到一个域名，比如 yahoo.com，会解析成一个 IP 地址，比如 206.190.36.45。GeoDNS 从用户视角看也是这样，然而，GeoDNS 会基于用户的地理位置返回不同的 IP 地址。一个欧洲的用户和一个澳洲的用户得到的 IP 可能不同，结果是每个用户都会访问到距他最近的一个 Web 服务器。GeoDNS 的目标就是将用户分发到离他最近的数据中心进而实现最小的网络延迟。

基础设施层面的另一个扩展是在全球范围部署若干边缘缓存（edge-cache）服务器，进一步减少网络延迟。边缘缓存服务器的使用依赖于应用的自然特性。边缘缓存服务器最高效的用法是像反向代理服务器那样缓存整个页面。当然，边缘缓存服务器也会提供其他服务。

边缘缓存是一种距离用户较近的 HTTP 缓存服务器，便于部分缓存用户的 HTTP 流量。从用户浏览器发起的请求到达边缘缓存服务器，边缘缓存服务器决定从缓存中直接返回响应页面，还是通过发送背景请求到 Web 服务器获取响应页面的其他部分最后进行组合。边缘缓存服务器还可以决定某个页面是不可缓存的，也可以决定是否可以代理整个 Web 服务。边缘缓存服务器可以缓存整个页面也可以缓存页面片段。

图 1-9 所示为根据用户全球地理位置提供不同请求服务的多数据中心高层架构图。在这种场景中，位于欧洲的某个用户进行域名解析时将会得到一个位于欧洲的边缘服务器的 IP 地址。然后，他就可以从这个边缘服务器或者某个应用服务器获得请求响应。他也将从 CDN 供应商那里下载静态文件，比如 CSS 或者 JavaScript 文件，由于大多数的 CDN 供应商会在多个国家建设数据中心，这些文件也会从距离用户最近的数据中心获取。类似地，位于北美的用户将会访问北美的边缘缓存服务器从北美的 CDN 数据中心下载。随着你的应用未来持续发展，你也许会考虑将主数据中心切分成多个数据中心并把它们部署到离用户较近的位置。通过将应用和数据存储部署到离用户较近位置的方式，可以实现更短的访问延迟和更少的网络成本。

现在，让我们讨论更大范围的应用生态系统和更高层面的基础架构设施，单个数据中心如何支撑可伸缩性。



数据中心基础设施架构概览

我们回过头来看看现代 Web 应用的各种技术。接着前面小节，我们进一步深入探讨这些贯穿全书的主题，不过首先我要给出一个通信流程和各种技术类型的功能全貌。

图 1-10 所示为一个通信流程的概览图。它展示了从用户终端设备到基础架构各个层的通信流程，可以说是本书最重要的图之一，展示了设计实现一个可伸缩 Web 应用需要熟悉的全部核心组件。你也可以认为这是一张导览图，我们将会在后面不同章节深入到不同的部分中去。事实上，本书的结构设计和数据中心的架构非常相似，图中的每一个部分都可以对应到书中的不同章节。

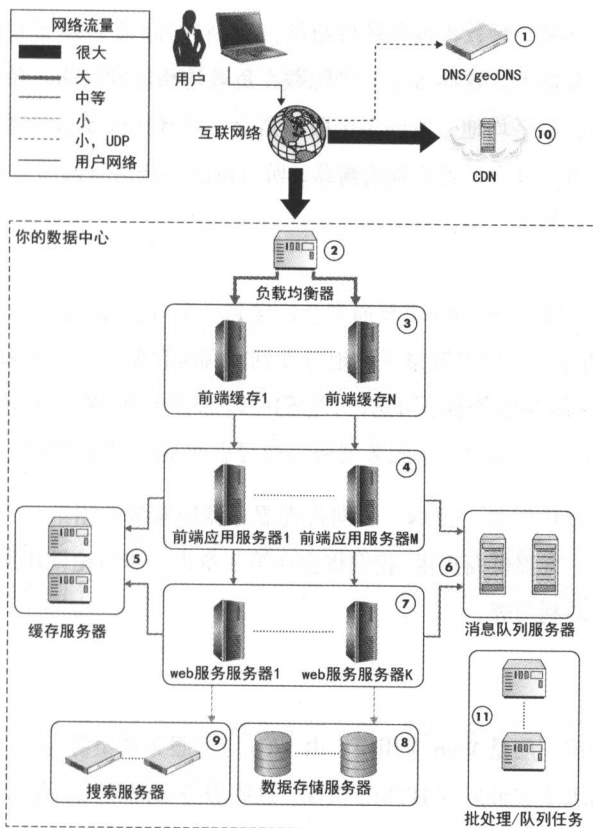


图 1-10 数据中心基础架构高层概览

图 1-10 中展示的许多组件都服务于某个特定的功能，能够被独立增加或者移除。不过，更常见的是在一个大型 Web 应用中可以看到所有这些组件。

前端

前端是我们应用栈上的第一个部分，包含了直接与用户设备交互的一系列组件。前端部分可能是我们数据中心内的，也可能在我们数据中心之外，这要看部署细节和第三方服务的使用情况。这些组件不包含任何业务逻辑，主要目的是提升系统处理能力和伸缩能力。

从图 1-10 的顶端看去，用户请求先到达 GeoDNS 服务器进行域名解析。DNS 决定哪个数据中心距用户最近，然后返回给用户这个数据中心的负载均衡器②IP 地址。

负载均衡器是一种软件或者硬件组件,可以将访问某个 IP 地址的访问流量分发到多个服务器上,这些服务器则隐藏在负载均衡器的后面。负载均衡器可以将用户访问负载平均地分发到多个服务器上,并且允许动态地增加或者移除服务器。由于用户只能看到负载均衡器,所以可以在任何时候添加新的 Web 服务器而无须停止服务。

从互联网上过来的 Web 访问流量通常会连接到一个有较强硬件能力的负载均衡器的 IP 地址。负载均衡器会把请求流量平均地分发到前端缓存服务器③或者直接分发给前端应用服务器④。前端缓存服务器是可选的,它们可以部署在距数据中心较远的地方或者直接跳过。在某些情况下,前端缓存服务器可以减轻后面基础架构设施的负载压力。

一般说来,负载均衡器、CDN、反向代理服务器通常都使用第三方服务,有时候,这个前端层全由第三方提供商提供。我们将会在第 3 章进一步讨论使用第三方提供商前端组件带来的伸缩性的利与弊。

Web 应用层

整个应用栈的第二层是 Web 应用层,由 Web 应用服务器集群④构成,主要职责是处理用户的 HTTP 请求生成最后的 HTML 页面。这些服务器通常用轻量级的 Web 开发框架(PHP、Java、Ruby、Groovy 等)构建,实现最小的业务逻辑,主要任务是生成用户界面。整个 Web 应用层的主要用途是处理用户交互并转换成内部服务调用。这个 Web 应用层越简单功能越少,整个系统越好。复杂的业务逻辑应该在 Web 服务层完成,实现更好的复用及减少需求变更,因为展示层的变更是最频繁的。

由于 Web 应用服务器被设计成完全地无状态,因此很容易实现伸缩性。对于一个完全无状态的服务器,实现伸缩性只需要简单地通过负载均衡器向服务器池子添加服务器就可以了。第 3 章中将讨论前端层和 Web 应用层。

Web 服务层

应用栈的第三层是 Web 服务层,由各种 Web 服务组成。这是非常关键的一个层,包含了最主要的应用逻辑。我们保持前端服务器简单并且尽量少包含业务逻辑,目的就是把展示层从业务逻辑中分离出来。通过创建 Web 服务层,我们更容易实现功能分割。我

们可以对某个功能实现专门的 Web 服务，然后独立地对它进行伸缩。例如，对于一个电子商务 Web 应用，可以开发一个产品类目服务和一个用户信息服务，每个服务都是不同的功能类型，都有完全不同的伸缩性需求。

前端应用和 Web 服务之间的通信协议通常使用基于 HTTP 的 REST（表述性状态传递）或者 SOAP（简单对象访问协议）。依赖于这种实现，Web 服务可以相对简单地进行伸缩。由于我们保持这些服务无状态，水平伸缩可以通过向服务器池添加服务器实现，相对应的，数据层实现伸缩性则困难得多。

近些年来，Web 应用间集成变得越来越流行，常见的实践是将 Web 服务暴露给第三方或者直接给用户。这就是 Web 服务经常和前端应用服务器平行部署，而不是部署在前端之后的原因。

第 4 章将详细讨论 Web 服务。至于现在，我们只需要先了解 Web 服务是我们应用的核心，是将功能分割到不同子系统的一种方式，允许独立开发进行独立伸缩。

附加组件

由于前端服务器④和 Web 服务⑥都是无状态的，所以 Web 应用需要部署一些附加的组件，比如对象缓存⑤和消息队列⑥。

对象缓存服务器既在前端服务器使用，也在 Web 服务中使用，主要目的是减轻数据存储服务器的负载压力及通过对部分数据进行预先计算实现响应加速。缓存服务器的细节描述将在第 6 章讨论。

消息队列被用来将某些处理延迟到稍后的阶段处理，并且将处理操作委托给消息处理者服务器⑪。发送给消息队列的消息通常来源于前端应用及 Web 服务，然后这些消息被特定的消息处理者机器处理。有时候，Web 应用会有一个批处理服务器集群或者若干定时运行的作业（由定时执行任务控制）。这些机器⑪不会去响应用户请求；它们是离线的作业处理服务器，提供类似异步通知、订单处理，以及其他一些允许较高延迟的功能。消息队列及消息处理者的内容将在第 7 章中进一步讨论。

数据持久层

最后，我们来到数据持久层⑧和⑨。一般说来，这是最难以进行水平伸缩的一层，

所以我们会花费大量篇幅讨论各种关于持久层的伸缩策略及水平伸缩方案。这也是大数据和 NoSQL 这些新技术快速发展的领域，这些技术可以存储和处理快速增长的海量数据，无须关注这些数据的来源和格式。

在过去的 10 年里，数据持久层技术变得越来越引人入胜，过去那种 SQL 数据库一统天下的局面一去不复返。正如马丁·富勒所言，这是一个持久层技术百花齐放的时代，同一家公司可能会选择使用多种数据存储方式以满足公司业务并实现更好的可伸缩性。第 5 章和第 8 章将进一步讨论这些技术。

在过去的 5 年里，搜索引擎由于其丰富的特性和有很好的开源项目支撑，变得越来越流行。由于搜索引擎和其他数据持久存储相比有很多不太一样的特性，所以图 1-10 中将它当作一个独立的组件画出来，相信熟悉这种技术还是很有必要的。

数据中心基础架构

我们从单一服务器架构开始，经过好几次复杂的架构变迁，基础架构也拥有了很多不同的平台，可以在多台服务器上分摊负载。图 1-10 中每种组件都会负责各自的功能并帮助应用获得很好的伸缩性以应对百万级的用户访问。

图 1-10 中各种组件的布局是经过深思熟虑的，主要目的是帮助那些相对比较慢的组件降低负载访问压力。可以看到，到达负载均衡器的访问流量被分成多份分发给前端缓存服务器。由于有些请求响应是被缓存了的，所以访问流量被减少，只有部分请求流量到达前端服务器④。应用层缓存⑤和消息队列⑥进一步减少访问流量，只有很少的访问流量会到达后端 Web 服务⑦，而 Web 服务可以使用消息队列和缓存服务器进一步减低访问流量。最后，只有在非常必要的情况下，Web 服务层才会连接搜索引擎及主数据存储去读写必要的信息。通过在数据层之上增加各种相对比较容易实现伸缩性的处理层，可以以一种低成本的方式实现整个系统的可伸缩性。

有一点需要特别注意，就是没有必要为了伸缩性实现图 1-10 中的所有组件。相反，尽可能少地使用不同种类的技术，因为每增加一种技术，就是在增加系统的复杂度，也是在增加维护的成本。使用很多不同的技术看起来很酷，但是却让发布、维护、调试变得更困难。如果你的应用只是需要一个简单的搜索功能，也许只要一个前端服务器和一个搜索引擎集群就能够满足你所有的伸缩性需求。如果你能增加服务器实现现有的每一层的伸缩性，并且也能满足你全部的业务需求，为什么还要不厌其烦地使用各种额外的组件呢？将

来我们讨论各个组件细节的时候，我们还会继续回过头来看图 1-10 的。

应用架构概览

目前为止，我们讨论了基础架构和伸缩性演化的各种阶段。现在让我们看看应用本身的高层次架构。

应用架构不是关于某个框架或者某个特定技术的，也不是关于 Java、PHP、PostgreSQL 或者数据库表结构的。应用架构是关于业务模型的演化。市面上已经有很多不错的关于领域驱动设计和软件架构的书，这些书可以帮你更好地熟悉软件设计的最佳实践^[1~3]。遵循这些最佳实践，我们将业务逻辑放在应用架构的核心位置。归根结底，这些业务需求驱动着其他各种决策。没有正确的领域模型和正确的业务逻辑，我们的数据库、消息队列及 Web 框架都没有任何意义。

不管应用是一个社交网站，还是一个药品服务，甚至是一个赌博 APP，它总归是有某些业务需要一个领域模型的。通过将这个模型放到应用架构的核心，我们确保各种组件围绕这个核心展开，服务于这个业务，而不是其他什么东西。如果把技术放在核心位置，也许我们会得到一个很赞的 Rails 应用，但是不太可能得到一个很棒的药品服务应用。

领域模型表示一个应用的核心功能，重点在于业务而不是技术。领域模型解释了关键术语、角色和操作，而不关心技术实现。一个自动柜员机（ATM）的领域模型的关键词是：现金、账户、负债、信用、身份认证、安全策略等。同时，领域模型不关心硬件和软件实现。领域模型是关于应用要解决的业务问题的本质描述。

图 1-11 所示为应用组件的简单布局。这里已经假设用户在单一应用中使用我们的系统，但是在系统内部，应用被分解成多个（高度自治的）Web 服务。

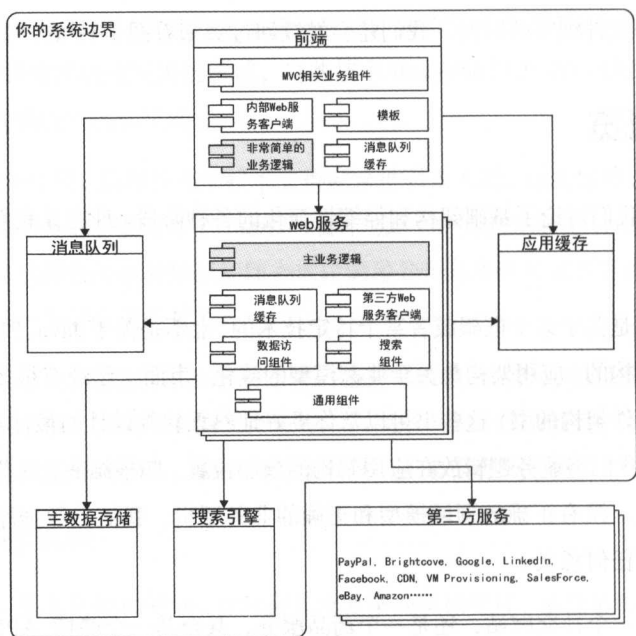


图 1-11 应用架构高层概览

让我们在接下来的几个小节中详细讨论图 1-11 中的各个部分。

前端

跟我们讨论基础设施架构图一样，我们从图 1-11 的顶端看去，这也是处理用户请求的起始点。需要注意的是，应用架构的核心是主要业务逻辑，但是简化起见，我们从前端组件看去。

前端的主要职责是成为用户的接口，用户通过网页、移动 APP 或者 Web 服务调用完成和应用的交互。无论实际交互方式是什么，前端应该是介于公开接口和内部服务调用之间的处理层。前端可以被视作是应用的“皮肤”或者应用的插件，是系统向用户呈现的功能展示，因此不应该是整个系统的核心或者重点。一般说来，前端应该尽可能地保持简单。

通过保持前端简单，我们可以复用更多的业务逻辑。由于业务逻辑只存在于 Web 服务层，因此我们可以避免视图和业务强耦合带来的问题。这样我们也可以实现前端的独立伸缩，使前端不需要处理复杂的逻辑及记录各种业务数据，只专注于处理较高的并发访问请求。

前端代码和模板及我们选择的 Web 框架（比如 Spring、Rails、Symfony）紧密耦合，主要制约因素是用户接口、用户体验需求，以及使用的 Web 技术。前端应用开发建立在 HTTP 通信之上，包括 AJAX 及 Web Session。把服务隐藏在前端之后，保持服务的简单与职责单一，使服务只关注业务逻辑而不是视图呈现与特定的 Web 开发技术。

模板、Web 流程、AJAX 全都是具体而特定的问题，将这些问题与业务逻辑隔离可以让前端发展得更快速更独立。把前端当作系统内部独立的应用进行开发还会带来另一个好处：我们可以使用不同的技术栈进行开发。开发 Web 服务与开发前端应用使用不同的技术并非毫无道理。举个例子，你可以使用 Groovy、PHP 或者 Ruby 开发前端，而使用纯 Java 开发 Web 服务，各有优势。

提示

你可以认为前端是一个可移除的插件，可以用不同语言重写、可以插入各种类型的前端。你可以移除一个基于 HTTP 的前端而插入一个移动应用的前端或者一个命令行前端。对前端的这种看法和态度可以让你有更多选择，可以确保让你的前端和核心业务逻辑分离。

前端不应该关注数据库或第三方服务。允许前端代码出现业务逻辑会导致代码难以复用及系统更高的复杂度而难以维护。

最后，可以允许前端组件发送事件消息给消息队列，以及允许使用缓存后端，消息队列和缓存对于提升性能和改善伸缩性都是很重要的手段。无论我们缓存整个 HTML 页面还是 HTML 片段，都会比在构建 HTML 时缓存数据库查询更节约处理时间。

Web 服务

“SOA 就像雪花，不会有两个完全相同。”

——David Linthicum

Web 服务是处理主要流程和实现业务逻辑的地方。如图 1-11 所示，Web 服务在整个应用架构中处于中心位置。这种架构通常被称为面向服务的体系架构（SOA）。不幸的是，SOA 这个词现在被用烂了，当你和不同的人交流的时候，你会得到一个完全不同的定义。

SOA 是一种以低耦合和高度自治的服务为中心的软件架构，主要目标是实现业务需求。SOA 倾向于所有的服务都基于约定有清晰的定义，并且都使用相同的通信协议。在 SOA 的定义中，我不太关注 SOAP、REST、JSON 或者 XML，这都是实现上的细枝末节而已。不论你使用什么技术或者什么协议，只要你的服务是松耦合的并解决了一组特定的细分领域的业务需求就可以了。第 2 章将解释耦合与最佳设计原则。

提示

看一下两个相似的术语：SOA 与 SOAP（简单对象访问协议）。虽然这两个词经常被看到在一起，不过 SOA 是一种架构风格，而 SOAP 则是一组定义、发现、使用 Web 服务的技术。你可以实现 SOA 而不使用 SOAP，也可以在其他架构风格中使用 SOAP。

我建议你阅读本书最后的推荐文献^[31, 33, 20]从而对 SOA 有更多了解，但是需要注意，SOA 不是所有问题的唯一答案，还有其他一些架构：比如分层架构、六角架构和事件驱动架构。你可以在其他系统中看到这些架构的应用。

分层架构是一种将不同功能划分到不同层次的架构。低层的组件暴露一组应用编程接口（API）给高层组件供其调用，不过低层组件永远不会依赖高层组件提供的功能。分层架构的一个例子是操作系统及其各种组件，如图 1-12 所示，包含硬件、设备驱动、操作系统内核、操作系统库、第三方库、第三方应用等层次。每一层都消费其低层提供的服务，但是低层永远不会消费上层提供的服务。另一个比较好的例子是 TCP/IP 编程栈，每一层都依赖底层提供的协议并增加新的功能。

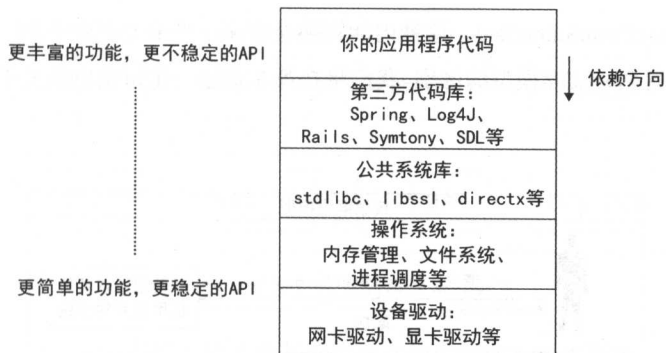


图 1-12 分层架构示意图

分层可以强制结构化并减少耦合，低层组件变得更简单和系统，其他部分更少耦合。我们也可以替换低层组件只要实现相同 API 就可以了。分层架构的一个重要影响方面是越到底层稳定性越强。你可以相对随意地变更高层组件的 API，因为依赖它们的东西很少。但是，变更低层 API 就代价不菲了，因为有大量代码依赖这些已经存在的 API。

六角架构认为业务逻辑是架构的中心，所有数据存储、客户端、其他系统之间的交互都是平等的。业务逻辑和每一个非业务逻辑组件之间都有一个约定，但是没有底层和顶层的划分。

在六角架构中，用户和应用的交互与应用和数据库系统的交互没有区别。它们都存在于应用业务逻辑之外而且都遵循一个严格的约定。定义好这些边界，你可以用一个自动化测试驱动代替一个人或用某个存储引擎代替数据库系统而不会对系统核心造成任何影响。

事件驱动架构 (EDA)，简单地说，是以一种不同的方式去思考行动，顾名思义，即对已经发生的事件做出反应。传统架构对请求做出响应并完成请求的工作，传统编程模型中，我们认为我们是请求某项工作要完成的人，例如，`createUserAccount()`，我们期望在我们等待结果的过程中，这个操作会被执行完成，然后我们继续后面的过程。在事件驱动模型中，我们不会去等待事情被做完。无论什么时候，我们和其他组件交互，我们只是宣布某件事情已经发生然后就处理后面的过程了。类比前面那个例子，我们可以宣布一个事件

UserAccountFormSubmitted。这种内涵转换会带来一些有意思的东西。图 1-13 所示为了这两种交互模型的区别。我们将在第 7 章进一步讨论更多关于 EDA 的细节。

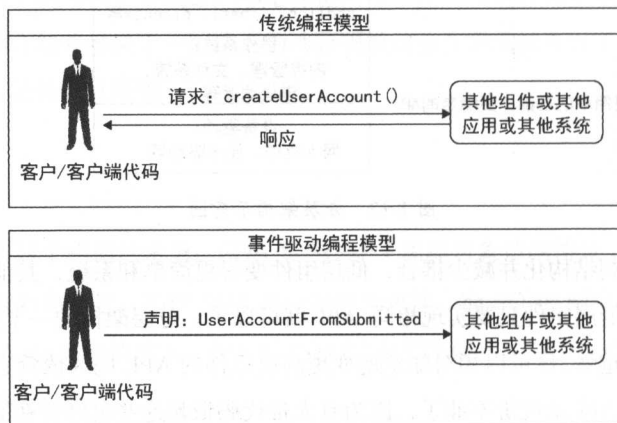


图 1-13 传统编程与事件驱动编程对比

无论实际系统使用什么样的架构风格,所有架构的目的都是将系统切分成更小的独立的功能单位。这样做的目的是构建更高层次的抽象以实现隐藏复杂性、减少依赖、各部分独立伸缩,以及每个部分并行开发。

提示

可以认为 Web 服务层由若干高度自治的应用组成,每个 Web 服务自己就是一个应用。Web 服务也可以彼此依赖,不过说到底还是少互相依赖为好。Web 服务提供一个更高层次的抽象,可以让你看清整个系统并理解它。每个服务都隐藏了自身实现的细节并呈现一个简化的高层次的 API。

理想情况下,所有的 Web 服务都是完全独立的。图 1-14 所示为一个假想的电子商务网站平台的各种 Web 服务。例如,文本分析服务是一个独立的服务,可以仅仅基于文本内容分析文章大意。如果每个服务都不需要用户数据或者其他服务的支持,它就是完全独立的。

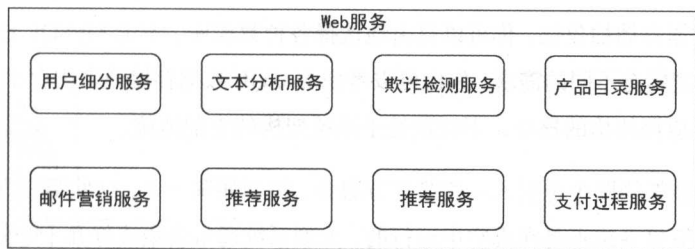


图 1-14 Web 服务层的服务概念视图

不幸的是，像上面这种所有服务都是完全独立的可能性几乎不存在。大多数情况下，服务之间多少会有一些依赖。例如，一个客户细分服务可能基于用户活动及社交网络数据产生的客户画像之上。为了将用户划分到不同分类，我们需要将这个服务和用户数据、用户活动历史、第三方服务集成起来。用户细分服务显然比文本分析服务更需要依赖其他服务。

不论你的 Web 服务是如何实现的，不要忘了它们的主要目标：解决业务需求。

支撑技术

如图 1-11 所示，Web 服务周边还有一些其他更小的方框，分别写着消息队列、应用缓存、主数据存储、搜索引擎等。这些方框被隔离开了是因为它们通常用一些其他技术实现，一般都是一些第三方的软件产品，通过配置和我们的系统集成起来。由于都是一些第三方的技术，所以在架构图中用一些空白的方框画出。

你可能已经注意到，数据库（主数据存储）也只是一个位于图中左下角的简单空白方框。这是因为数据存储仅仅是一种技术，是实现细节而已。从应用架构的视角看，数据存储是一个让我们读写数据的地方，我们并不关心需要多少服务器，如何进行伸缩，怎样进行数据复制及容灾，甚至如何存储数据。

提示

对待数据存储应该和对待缓存、搜索引擎、消息队列一样，只是一种即插即用的扩展组件。如果你决定更换持久层存储或者更换缓存后端，你应该做到只更换数据连接组件就可以，保证整个架构不受影响。

通过将数据存储抽象化，你可以自由地选择各种数据库，不论 MySQL 还是其他数据库。如果应用逻辑有不同的需求，你也可以考虑 NoSQL 数据存储或者内存型存储。记住，数据存储不是应用架构的核心，不应该处于系统架构的支配地位。

最后，我在架构图中也引入一些第三方服务，特别强调一下，这些第三方服务也很重要。我们的计算机系统不是在真空中运行的，大型系统通常会集成好几十个外部系统，严重依赖这些系统提供的功能。第三方服务在我们的控制之外，处于我们系统边界之外。因为我们不能控制它们，我们也就不能指望它们一直运作正常、没有 BUG、像我们期望的那样快速伸缩。通过设计一个间接的访问层将第三方服务和我们的系统隔离开来，可以最大程度地降低使用风险和系统依赖。

小结

架构是从软件设计师的视角看，基础设施是从系统工程师的视角看。每一种视角都是从不同方面展示同一个问题：如何构建可伸缩的软件。读完本章后，你应该能够画一幅图，表现架构和基础设施如何统一在一起构建一个可伸缩的 Web 应用。当我们深入到各个组件细节中的时候，这张图将会非常重要。

就像你看到的那样，伸缩性不是一个容易搞定的问题，涉及软件与架构设计的方方面面，需要广泛用到各方面的技术知识。你需要明白各种技术组件如何组合在一起，它们各自的作用是什么，它们的优缺点是什么，才能真正驯服伸缩性这头猛兽。要设计一个可伸缩的 Web 应用的关键是把握好架构、基础设施、技术、算法、业务需求之间的冲突。第 2 章中我们将讨论软件设计的原则，这也是可伸缩应用的预备知识。

2

软件设计原则

许多实际项目中遇到的伸缩性问题其实可以归因于违反了软件设计的核心原则。软件设计原则比伸缩性更抽象更通用,软件设计原则能够为构建可伸缩的软件奠定一个坚实的基础。

本章中提到的设计原则有些是关于面向对象设计的,有些是直接和伸缩性相关的,不过这些设计原则一般都很抽象并具有很好的普适性,可以适用于各种场合。一个手艺娴熟的软件工程师应该非常熟悉好的和差的设计实践,了解隐藏在设计决策背后的那些驱动要素。现在就让我们一起了解这些设计决策。

简单

“使事物尽可能简单,但是不要过于简单。”

——阿尔伯特·爱因斯坦

最重要的原则就是使事物简单。简单是你的北斗星、是你的指南针、是你的远期承诺,但是使软件简单还是相当困难的,因为软件天然就是错综复杂的。关于简单也没有一个衡量标准,当你判断一个东西是不是过于简单的时候,你首先要回答的是对准而言及对什么时候而言。例如,是对你而言还是对客户而言过于简单?是对现在开发而言还是对将来维

护而言过于简单？

简单不是走捷径，不是为手边的问题找一个最快的方案。简单是让别的软件工程师以一种最容易的方式使用你的方案；简单是当系统变得更庞大更复杂的时候依然能够被理解。学习如何简单最好的途径是经验，特别是当你开发了很多不同的应用，使用很多不同的框架和语言之后。重温你曾经写过的代码，区分复杂度，以此寻找简单的方案。从你自己的错误开始是学习的第一步。随着时间的推移，你会具有敏锐的感觉和能力，从而快速判断出长期而言什么是更简单的方案。如果有机会能够找到一个导师或者能够和擅长发现简单的人一起共事，你会进步得更快。简化你的产品可以从以下 4 个基本步骤开始。

隐藏复杂与构建抽象

隐藏复杂与构建抽象是实现简化最好的方法之一。随着系统的发展，你会发现你已经无法了解整个系统的全部情况了，因为系统已经变得非常庞大。人类的大脑处理能力也是有限的，要么对系统整体有个了解而不知道细节，要么知道系统的一小部分细节而不了解整体。保持软件简单可以让你收放自如地从整体到细节各种粒度地查看系统。不过，如果系统很庞大，是无法保持整体简单的，你能做的只是保持局部简单。

达到局部简单的主要方式是确保在设计和实现两个方面上，任何单个的类、模块、应用的设计目标及工作原理都能被快速理解。当你看到一个类的时候，你应该能够快速理解它的工作原理而无须知道系统其他部分的全部细节。你需要做到只看着这个类就明白这个类能干什么。当你看一个模块的时候，你能不看这个模块本身，而把这个模块当作一大堆类来看，而每一个类都是可理解的。如果再缩小了看，当你看一个应用的时候，你能一眼就看出那些关键的模块和它们的主要功能，而无须知道模块里的类的细节。最后，当你看整个系统的时候，你能快速看出顶层的应用和它们的职责而无须知道每个应用是如何运行的。

我们来看一个抽象的例子，如图 2-1 所示，圆圈表示类或者接口，边表示类或接口的依赖关系。在这里，复杂度不是指这个网络中有多少个节点，而是指这些节点之间有多少个边，节点表示类，而边表示类之间的依赖关系。好的设计原则是类之间的依赖关系尽量少。

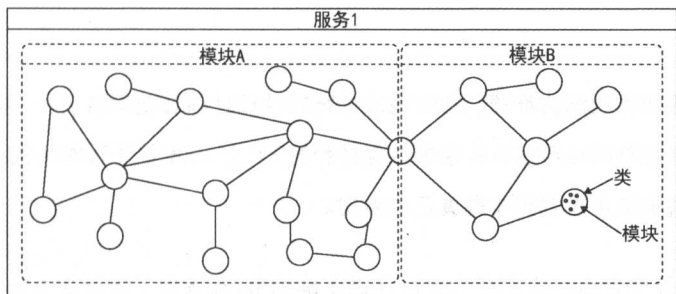


图 2-1 抽象的级别

为了实现局部简单，需要将不同功能分割到不同的模块中，这样当你从一个比较高的抽象层次去观察应用时，无须操心每个模块的职责是什么，只需考虑这些模块是如何交互的就可以了。如果你从更高的层面去看图 2-1，即模块的层面，你可以忽略模块内部的细节，只关注模块间的交换即可。在这个例子中，模块 A 和模块 B 之间的交互只有一个两边都可见的公共接口。

在庞大又复杂的系统中，当你创建一个独立服务的时候，需要添加一个抽象层。服务暴露这个更高层次的抽象，实现这些抽象所承诺的功能，从而隐藏其复杂性。

避免过度设计

实现简化的第二个实践是避免过度设计。工程师喜欢挑战高难度的问题，喜欢开发一些复杂的软件。当你试图预测每一个可能的用例和那些极少发生的场景时，可能会忽视那些最主要的业务场景，会情不自禁地被那些想象出来的问题牵着鼻子走，不知不觉地就过度设计了，最后你会开发出一个比实际需求复杂得多的大而无当的系统。

好的设计方法是可以在后期逐渐添加新的功能特性，而不是一开始就开发一个超级大的系统。早期先构建一个合理的抽象层次，然后迭代地增加新特性，要比一开始就设计好全部的功能为将来各种可能进行开发好得多。

众所周知，Java 社区格外喜欢在哪怕最小的地方也要过度设计。幸运的是，最近几年，Spring 这类开发框架及基于 Java 虚拟机的动态语言（比如 Groovy）逐步将开发带入正确的方向。开发工程师们开始注意起简化和常识，而不是企图开发一个没有人用也没有人懂的想象中的系统。

提示

如果你喜欢解决难题，那么每次进行软件设计前都要问自己：“这个设计是否可以更简单并且可以在将来依然保持弹性？”。开发易于理解的简单系统并能证明其将来是可扩展的才是真正的难题。

过度设计经常发生在人们以为自己在做正确的事，却选择了错误的方向或者太过关注未来的需求。我建议你问问自己“这里我需要如何做出权衡？”或“这里我是否真的需要？”，也建议你与业务相关方多接触，尽量去了解什么是最大的风险及还有什么没有搞清楚。否则，你会按照那些没用的教条花费大量的时间去构建一个没人用的系统。本章的大部分设计原则都要花费一定功夫才能达成，你需要搞清楚合理的复杂和过度设计之间的界限。要做到这一点其实非常困难，因为这个界限不是那么的黑白分明易于寻找，而是需要在一大片灰色地带中自己斟酌权衡。

尝试测试驱动开发

接受测试驱动开发（TDD）的方法论也可以提高简化性。你不必在所有地方都搞 TDD——在某些方法上搞 TDD 就可以获得一个全新的视角。

测试驱动开发是一种开发实践：工程师先写测试代码，然后写功能实现代码。这是一种基本的方法，值得体验。其最主要的好处是没有单元测试就没有代码，所以也就没有无用的代码。由于开发者先写测试，他们就不会写那些没必要的代码，否则他们就又要去写测试代码。另外，单元测试可以被当作某种文档，它们可以展示代码的实现意图、具体用法、期望的执行结果等。

体验测试优先的方法论的另一个作用是：工程师可以更好地把握工作的重点。TDD 会强制工程师从用户的视角看待问题，有助于他们开发更清晰更简单的接口。因为要先写测试，所以必须要先想象如何使用你要开发的组件。为了写测试，你需要先假设用户是如何使用这个组件的，就不会只想着这个组件的内部实现细节。这种细微的差别会带来代码设计上的巨大提高和应用程序接口（API）的简化。

提示

无论你是否使用 TDD，都要从用户视角思考问题。假设公司新来的工程师需要使用你开发的接口，那么他想调用的方法是什么？他想传递的参数是什么？他期待返回的响应结果是什么？当你从这个视角去思考的时候，你就会更好地开发那些易于用户调用的代码。

从软件设计的简化范例中学习

如果复杂性处理得够好，反而难以体会软件简化的理念。如果事情总是顺其自然，如果可以毫不费力地理解系统，那么也许你遇到的是精雕细琢后的某种简化。认识到你使用的系统是良好设计的系统是一种非常赞的体验。无论何时你发现这一点，仔细分析它并寻找其中的模式。Grails、Hadoop、Google Maps API，都是一些简化做得很好的范例，值得认真研究。试着分析这些框架：

• Grails

Grails 是 Groovy 语言上的一个 Web 框架，类似于 Rails (Ruby 语言的 Web 框架)。Grails 是一个简化如何变得透明的极好的例子。随着研究这个框架的深入并使用它，你会意识到这里的每一样东西都被仔细考虑过。你会看到事情总是和期望的一样，扩展功能总是毫不费力。你会意识到这个框架变得更简单是不可想象的。Grails 是一件让开发变得容易的杰作。请阅读 *Grails in Action*^[22] 及 *Spring Recipes*^[14] 去学习更多相关知识。

• Hadoop

熟悉 MapReduce 编程范式及 Hadoop 平台。Hadoop 是开源技术领域的杰作，可以处理 PB 级的数据。Hadoop 是一个庞大而复杂的平台，但是它很好地隐藏了其复杂的实现。开发必须要掌握的是一个极其简单的编程 API。当你进一步深入学习 Hadoop 的时候，才会发现它解决了多少困难的问题，才让开发者处理几乎无限的数据变得如此简单。如果想对 MapReduce 和 Hadoop 有个基本的了解，推荐阅读 MapReduce 原始论文^[w1] 及 *Hadoop in Action*^[23]。

• Google Maps API

浏览 Google Maps API，有一些我非常欣赏的 API。这些年来，这些 API 总是在变

化，但是它们一直都以一种格外简单的方式使这些 API 在解决复杂问题的同时保持足够的弹性。如果你只是需要一个地图并做一个标记，那么你只需要在一个小时内就搞定全部，包括创建 API keys。当你进一步深入研究的时候，你会发现更多有趣的特性，比如地图覆盖、用户接口（UI）定制、地图式样，以及空间自适应。

当你读完本章，你会看到更多改善简单性的设计原则。简单对你的系统的伸缩性有一些潜在的价值。没有简单性，工程师就没法理解代码，不能理解软件，系统就无法保证持续发展。一定要记住，对于伸缩性，最好是设计简单的系统然后让它好好运行，而不是设计复杂的系统然后突然崩溃了。

低耦合

第二个重要的设计原则是保持系统各个部分之间尽量低耦合。

耦合用来度量两个组件之间互相关联及依赖的程度。耦合度越高，依赖越强。低耦合是指两个组件之间只有必要的一点点关联和依赖，而没有耦合则是指两个组件之间完全不感知对方的存在。

保持系统低耦合对于系统的健康及伸缩性至关重要，它甚至会影响团队的士气。让我们看一下低耦合和高耦合的不同影响。

- 高耦合意味着任何一行代码的改动都需要你检查系统各个部分是否存在问题。耦合度越高，各种出乎意料的依赖越多，引入 BUG 的机会就越高。想象一下，你对用户认证处理做了一点改动，然后你想起你还必须重构其他五个模块，因为这些模块都依赖了认证处理的内部实现。低耦合则可以让你改动代码而无须担心对系统其他部分造成麻烦。
- 低耦合可以保证复杂性局部化，即复杂只体现在模块内部，不影响整个系统。将系统分解成低耦合的多个部分，可以安排多个工程师分别独立开发各个部分。这样你就可以招募更多的工程师扩展你的团队，每个人都可以在不了解整个系统细节的情况下针对局部模块进行开发。

- 在更高层面上进行解耦合意味着将系统分成多个应用，每个应用都只关注相对较小的一部分功能。这样每个应用就可以按需分别伸缩。有些应用需要更多 CPU，而有些应用则需要更多 IO 吞吐能力或者更多内存。通过将系统解耦成不同部分，你可以针对具体应用特点提供相应的硬件配置以达到更好的伸缩性。

促进低耦合

促进低耦合的最重要的实践是小心地管理你的依赖。这个准则适用于类之间的依赖，模块之间的依赖，以及应用之间的依赖。

图 2-2 所示为一个系统内部类、模块、应用的布局关系。系统是全部——它包括了一切：你开发的所有应用和你系统环境中用到的所有软件。应用是系统内部最高层次的抽象，它们提供最高层次的应用服务。你也许会用到一个账户应用，一个资产管理应用，或者一个文件存储应用。

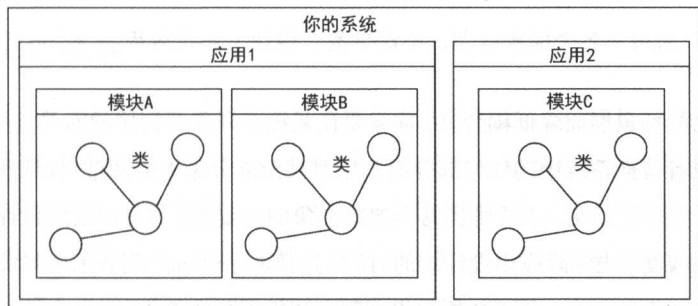


图 2-2 高耦合和松耦合的比较

在应用内部会有一个或者多个模块负责实现更精细更具体的功能特性。就像不同的应用通常是由不同的团队开发和部署，不同模块（比如信用卡处理、PDF 展现、FTP 接口）也应该对依赖它的开发团队保持足够独立，以便它们可以并行开发。如果你不够信任某个团队开发的某个特定模块，那最好不要让这个模块和你的应用耦合得太紧。

最后，模块内部包含了一些类，这是最小的抽象粒度。一个类应该只有一个目的而且代码最好不要超过两屏。关于单一职责原则本章后面还会继续讲到。

在面向对象编程语言中，比如 Java、C#、PHP，你可以通过使用 `public`、`protected`、`private` 关键字促进低耦合。你应该尽可能地将方法声明为 `private` 或者 `protected`。这样做的目的是越多方法声明为 `private`，对外界越隐藏自己的实现细节。隐藏细节是减少耦合和促进简化的一个非常好的手段。你的类越少访问其他类，你就越少需要关注这些类的工作原理。因为 `private` 方法只在类的内部被调用，所以这些方法可以被更容易地重构和修改，类的复杂性也被局限在类的内部，你不用到处查找这些方法在哪里被使用，是否会引起潜在的 BUG。相反，暴露太多 `public` 方法，就会增加外部代码使用它们的机会。一旦一个方法是 `public` 的，你就不知道它会在哪里被使用，你就不得不在整个应用中仔细查找那些调用它的地方。

提示

在写代码的时候，要吝啬一点。在满足需求的情况下，尽量少暴露内部信息和功能。暴露的信息越多越早，越增加耦合性，未来需求变更的时候越麻烦。这个法则也适用于各个抽象级别，不论是类、模块，还是应用。

要在较高的抽象层面降低耦合度，你需要让系统不同部分的接触面尽量少。低耦合可以让你重构或者替换系统中的任何部分而不用对其余部分做太多调整。找到解耦与过度设计之间的平衡是个艺术活，工程师经常会忽略抽象的重要性。你可以通过设计图来帮助自己做出正确的权衡。当你画应用设计图的时候，连接两个部分之间表示依赖关系的线条的数目决定了接触面的大小。图 2-3 所示为高耦合和松耦合的例子。

高耦合的应用很难只修改某个模块中的某个部分而不影响另一个模块。此外，模块之间彼此知道对方的内部结构并会直接访问这些部分。第二个例子展示模块具有更好的私密性。为了降低和外界的接触面，模块 B 的对外开放功能被隔离为一个很小的子集并被显式地声明为 `public`。另一件需要注意的事是第二个应用中模块间没有循环依赖。模块 A 可以被重构或者移除而不会影响到模块 B，因为模块 B 对模块 A 没有任何依赖。

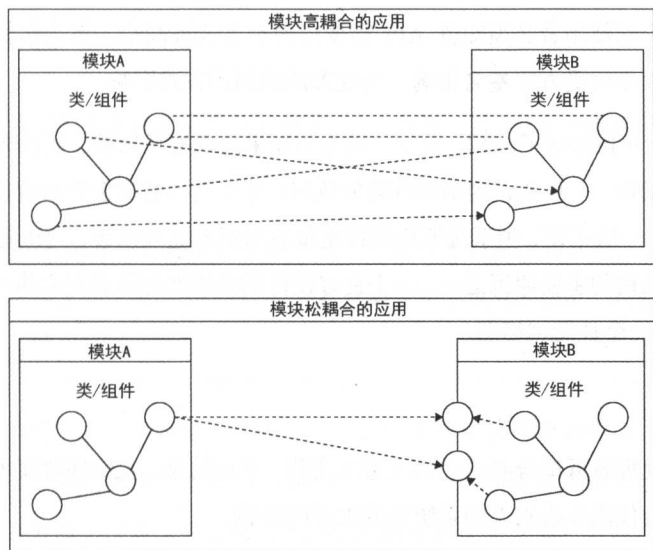


图 2-3 高耦合和松耦合的比较

避免不必要的耦合

在实践中，某些被广泛使用的方法事实上会增加耦合性。一个典型的导致不必要的耦合的例子是通过 `public` 的 `getter` 和 `setter` 方法暴露类的 `private` 成员变量。这种用法源于 Java Bean 的一些早期实践，当时提供 `getter` 和 `setter` 方法主要是为了适应代码操作工具及集成开发环境（IDE）。不幸的是，这种实践在整个 Java 社区范围内都被完全地误解了。代码和 IDE 集成这种不良的习惯也会影响到其他的语言 and 平台。

如果你用某种面向对象的语言进行开发，比如 Java 或者 PHP，创建一个新类的时候，不要给所有的类成员变量都加上 `public` 的 `getter` 和 `setter` 方法。这会破坏封装性增加耦合性。最好是一开始的时候全部方法都是 `private`，然后视情况将那些需要的方法改成 `protected` 或者 `public`。不同的编程语言会提供不同的方法去实现相同的目的——关键要记住尽可能多地隐藏而尽可能少地暴露。

另一个常见的不必要的耦合的例子是模块或者类的客户端代码必须按照特定的顺序进行方法调用。有时这样做是有合理的原因的，但是更多的时候是源于糟糕的 API 设计，比如需要调一个初始化方法。类及模块的客户端使用者应该不需要知道类的设计者期望他们如何使用这些类及模块。他们应该可以按照自己的方式任意调用类的 `public` 接口。如果

你要求你的客户端使用者必须知道 API 需要以何种方式被调用，那么你就是在增加耦合性，耦合接触面不但受方法签名影响，也受方法的调用方式影响。

最后，处于不同层次的应用、模块、类之间如果出现循环依赖，就意味着在设计耦合性方面比较糟糕。一旦意识到循环依赖的危害，那么这种情况很容易就能被注意到，也相对容易避免。一般来说，画系统架构设计图很容易暴露循环依赖，这也是我推荐在设计阶段一定要画架构的主要原因之一。一个良好设计的模块架构图看起来像一棵树（有向无环图），而不像一个社交网络图。

低耦合范式

要想很好地理解低耦合必须经历大量的实践。幸运的是，就像理解简单一样，你可以通过阅读别人的代码分析别人的系统获取大量的经验。

低耦合设计的一个很好的例子是 UNIX 命令行编程及管道（pipe）的用法。在 UNIX 系统中，无论何时创建一个进程，都会自动获得三个打开的文件进行读写操作，分别是标准输出、标准输入和标准异常。这些文件不必是实际的文件，它们可以是虚拟的文件句柄，指向终端、网络 socket、硬盘上的文件，或者是连接另一个进程的管道。UNIX 系统暴露一个非常简单的 API 去读写这些文件，你看到的仅仅是一些很简单的功能。操作系统将文件句柄的真实属性全部对编程隐藏掉了。无论是一个真实的文件还是一个网络 socket，程序都可以以同样的方式进行读写操作。这种设计允许 UNIX 命令行进行诸如 grep、sed、awk 及 sort 之类的编程去实现一个特定的功能，并且还能通过管道连接完成更复杂的任务。我个人认为 UNIX 文件操作是一个天才的设计，是利用“框架”实现程序间低耦合的一个极好的例子。

另一个比较好的低耦合的例子是 SLF4J。我强烈建议你了解 SLF4J 的结构，并且和 Log4J 及 Java Logging API 做个对比。SLF4J 的角色像是一个非直接层，把 log 接口调用和实现的复杂性分离开来，暴露了一组更简单更清晰更容易理解的 API。

低耦合是开发弹性软件最基本的原则之一。我建议你设计低耦合的模块放在一个更高的优先级上，也建议你去阅读一些从不同角度探讨低耦合的书籍。^[1, 2, 5, 10, 12, 14, 22, 27, 31]

不要重复自己 (DRY)

“我认为避免重复是最有价值的原则之一。只做一次，这是极限编程的格言。”

——Martin Fowler

重复自己意味着同样一件事你要做好几次。在软件工程生命周期里，你可能会在很多地方遇到重复，从重复的应用程序代码，到每次代码发布前重复的测试，到整个公司的运作。

提示

如果你把同样一件事做了一次又一次，你就是在浪费你的生命。不要把一件事做好几次，而是去做些比较酷的事，比如开发一些新特性或者为客户思考一些更好的解决方案。试着说服你所在的团队或者你的老板采取一些基本措施以避免重复——比如：如果有重复的代码那么代码审查不通过，每个新写的类都必须先写单元测试。

让开发工程师浪费时间做重复的事有很多原因。

- 采用低效的过程

这种情况在软件开发的各个阶段都会发生，设计新功能、交付、开会等，通过对这些地方进行持续改进可以获得良好的收益。获得反馈、持续改进、然后重复这个过程。很多时候，团队意识到自己在浪费时间却无可奈何。当你听到“我们就是这么干的”或者“我们一直都这么干”时，常常意味着这里有低效的过程并有改进的机会。

- 缺乏自动化

当你手工部署、编译、测试、打包、配置开发机器、准备服务器、为 API 写文档的时候，你就是在浪费时间。一开始，这些事看起来很简单，但是随着时间的推移，事情变得越来越复杂，花的时间越来越多。在你注意到这一点前，你可能会花费一整天的时间部署代码、测试发布，完全没有时间开发新功能。新增加的手工工作的工作量很容易就会被忽略，而巨大的浪费就是建立在这种不断增加的微小的工作量之上。从第一天开始就尝试将编译部署的工作自动化，当然你也需要

一直维护这些自动化的工具。

- **非得自己发明，也就是常说的重复发明轮子**

这个问题的表现症状是：不复用已有的代码而非要自己写代码。这种事常常发生在刚入行不久的年轻工程师身上，他们乐于重写那些很容易就能用到的东西（对于公司内部或者开源世界而言）。比如实现 hash 功能、排序、b 树、MVC 框架或者数据库抽象层。虽然这种重复不是字面意思上的重复，但这依然是在浪费时间，因为完全可以使用别人已经开发好的工具和代码库。任何时候，当我们想写一个通用类库的时候，都先上网搜一下，通常已经有很多实现得很好的开源代码了。

- **复制粘贴代码**

想象一下，你已经有一些代码，这份代码跟你想要开发的新功能很类似。为了节约时间，你把这坨代码复制粘贴过来，然后只对其中一小部分做了一点改动。祝贺你，你现在有两份相似的代码需要维护了。总会在某个时候，你会意识到你对这部分代码做的任何改动都不得不在其他地方再改一次，总会有一些已经修复的 BUG 会再一次出现，因为你忘了修改粘贴过去的那部分代码了。试着让团队遵守一些规则：比如“我们永远不会复制粘贴代码”。同时也给每个人权力在代码评审时指出重复的代码，让每个人彼此之间都感受一些良性的压力。

- **“这个代码不会用第二次所以就凑合一下”的态度**

有时候你要处理一些相对独立的问题，你会想“这些代码我不会用第二次，就随便搞搞好了”。有时，这些问题会再一次出现，你就不得不再用一次这些当初被当作一次性的随便搞搞的代码。那么问题来了，这些代码既没有文档，又没有单元测试，也没有合适的设计，很难用。更糟糕的是，其他工程师可能会复制粘贴这些随便搞搞的代码用在他自己的一次性脚本上，悲剧再一次上演。

复制粘贴代码

我相信复制粘贴代码是一个常见的问题，所以再多说一点。出现这种情况主要是因为开发工程师通常没有意识到自己写的代码越多，维护的成本和代价越大。复制粘贴导致应用的代码变得更多，更多的代码导致维护的成本更高，随着各种技术问题的堆积，这种成本会呈指数级上升。应用中存在重复代码，一个地方修改就需要所有重复的地方都修改，还要追踪复制的代码之间的差异，对所有复制粘贴的代码做回归测试。由于复杂度随代码

行数成指数级增长,所以复制粘贴的代价极其昂贵。事实上,复制粘贴代码是一个非常严重的问题,有人花费毕生精力去研究它。由 NASA (美国国家航空航天局) 发布的白皮书^[w2~w5]显示,有 10%~25% 的大型系统是代码复制粘贴的产物。

处理代码复制粘贴可能是一件让人很头痛的事,不过没有什么事是重构不能搞定的。一个好的开端是在代码库中搜索每一个重复的功能。一旦你这么做了,你就可以更好地了解哪些组件受到影响并且该如何重构它们,可以考虑创建一个抽象类或者将重复代码抽取到一个独立的、更通用的组件里。在对付重复代码的战斗中,组合或者继承都是你的朋友。

另一个对付复制粘贴代码的好办法是使用设计模式和共享类库。设计模式是解决一类通用问题的抽象方法,是软件设计层面的解决方案,可以应用到各种系统各种领域中。设计模式需要考虑如何组织面向对象代码、依赖、交互,但是不需要考虑具体的业务问题。设计模式会建议如何在模块中组织代码,但是不会指出需要使用什么算法或者业务特性及如何运作。设计模式超出本书的范围,你可以通过阅读推荐书单中的书籍^[1,7,10,36,18]掌握更多设计模式。

你也可以在更高层面上通过 Web 服务去对付代码复制。不要在每个应用中开发相同的功能,而是创建一个服务,然后在整个公司范围内复用这个服务。本书第 4 章将讨论更多有关服务的话题。

提示

阻止功能重复的一个办法是让功能使用变得最简单。例如,你的类库提供了 20 个功能,80% 的时间被使用的都只有其中的 5 个功能,那么保持这个 5 个功能尽可能地容易使用。容易使用的东西更容易被复用,如果使用你的类库是完成一件事情最容易的方式,那么每个人都乐意使用你的类库。如果使用你的类库很困难,那么大家就会想重复再搞一个。

基于约定编程

基于约定编程,或者说基于接口编程,是另一个重要的原则。基于约定编程是将客户端代码和功能提供者代码解耦合的主要方式。创建一个明确的约定,客户端仅仅看到这些

约定并只依赖这些约定编程。将系统不同部分解耦合并将变化隔离开会对开发带来很多好处，这一点在前面已经阐述过了。

约定是一组功能提供者承诺实现的规则。客户端代码可以理解这些规则并依赖它们。它们描述了哪些软件提供了哪些功能，但是不需要客户端代码知道这些功能是如何实现的。

“约定”这个词在不同场合代表不同意思。当我们讨论面向对象编程的成员方法时，约定是指方法的签名，定义了期望的输入参数和期望的返回结果。约定不会规定结果该如何计算出来，这是实现的细节，使用者无须关心这些。当我们讨论类的时候，约定是指类的 `public` 接口，包含了所有可访问的方法和签名。更进一步到抽象层，模块的约定包含了所有的公开可见的类/接口及它们的方法签名。如你所见，抽象层次越高，约定的复杂度和范围越广。最后，对整个应用而言，约定通常意味着一些描述 Web 服务 API 的表格。

我曾经提到，约定有助于将客户端代码和功能提供者代码解耦合。只需要保持按约定开发，客户端和功能提供者可以各自独立修改。这会让代码更隔离更简单。在设计代码的时候，尽可能创建明确的约定，也尽可能地依赖约定（而不是实现）进行开发。

图 2-4 所示为如何通过约定将客户端从功能提供者中分离出来。提供者 1 和提供者 2 是基于同样约定的两个可替代的实现。每个提供者都是独立的模块，由于它们都履行相同的约定，客户端可以使用其中任何一个而无须直接知道究竟使用的是哪一个。在客户端眼中，任何完成相同约定的代码都是同等的，这样进行重构、单元测试、修改实现都变得非常容易。

提示

要想让基于约定的编程更简单，可以考虑把约定当作真实的法律文件。当人们同意按照法律文件做事的时候，他们会变得对细节更敏感，因为如果某个条款没做到的话，他们就要承担相应的责任。在软件设计中也类似，低耦合的约定的每个部分都是在增加未来的责任。作为一个功能提供者，暴露太多不必要的东西就是在增加将来的负担，因为任何时候如果你想做出改变，你就不得和所有的客户针对改变的约定进行谈判（这个改变会在整个系统中蔓延）。

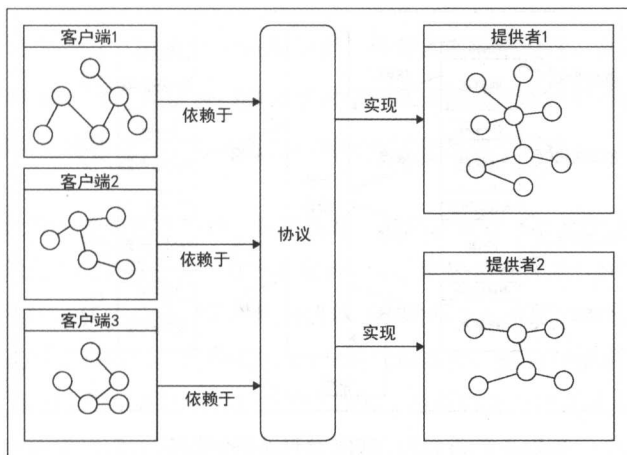


图 2-4 客户端与提供者解耦

设计类的时候,首先考虑的是客户端真正需要的功能是什么,然后设计一个最小的接口当作约定,最后实现代码以完成约定。针对类库和 Web 服务也遵循同样的方式。当你暴露一个 Web 服务 API 的时候,应该是明确的并要仔细考虑究竟暴露给客户端的是什么。需要的时候,应该做到容易增加新特性及发布更多数据,但是开始的时候应该尽可能地实现一个简单的约定。

为了展示基于约定编程的威力,我们来看一下超文本传输协议(HTTP)。HTTP 被不同系统使用不同语言在不同平台中实现,然而,HTTP 是最流行的协议之一。有些 HTTP 协议约定的客户端是 Web 浏览器,比如 Firefox、Chrome。它们被不同的组织以不同方式实现,按不同的进度更新和发布。另一方面,HTTP 提供者,主要是 Apache、IIS、Tomcat 这样的 Web 服务器,代码也被不同组织用不同的技术实现,并且被独立地部署在全世界各个地方。更棒的是,还有很多人们不知道的其他技术实现的 HTTP 约定,比如 Web 缓存服务器 Varnish 和 Squid,同时实现了客户端和提供者。图 2-5 所示为客户端和提供者如何通过 HTTP 协议松耦合。

尽管由于生态系统的复杂性,所有的应用都被互联网联系在一起,但 HTTP 还是体现出实现上的弹性及提供者透明的可替代性。HTTP 是通过约定进行解耦合的一个漂亮的例子,所有应用都有一个共同点,它们实现或者依赖一个相同的约定。

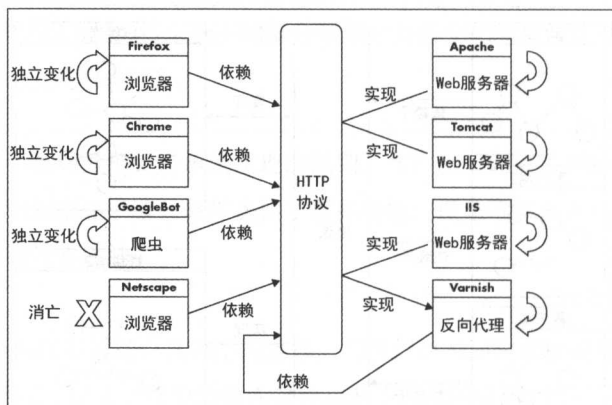


图 2-5 HTTP 协议松耦合实现

画架构图

“你知道真正的架构是什么吗？是一门画线的艺术。画一条线连接依赖并指明依赖的方向。”

——Robert C. Martin

对于架构师和技术领导者而言，画架构图是一项必备技能。一张架构图可以表达很多信息，所谓一图胜千言。通过架构图，你可以将系统设计文档化，跟其他人分享信息，也可以帮助自己更好地了解自己的设计。很多工程师不做代码的前期设计，会跳过画架构图这个阶段直接开始写代码。我看到过很多人这样做，我自己也经常这么搞。特别是适应了敏捷开发实践，学了很多创业方法学以后，不会花太多时间做前期设计，但是这并不意味着一直都不需要。

提示

如果你觉得画架构图挺难的，你可以尝试先画那些已经开发好的系统的架构图。你自己开发过的系统画架构图更容易一点，因为你毕竟更了解。等你熟悉了各种架构图的画法，再开始画前期设计架构图。从使用者的视角观察类接口的设计并进一步充实它们，试着为接口写单元测试，然后画一些简单的架构草图。通过假想用户视角及画简单架构图，你可以在写代码之前验证自己的设计并发现其

中的缺陷。等你熟悉了画各种架构图以后，尽量多做前期设计。如果你发现前期设计很难做的话，也不要沮丧，从先写代码变成先做设计本来就不是一件容易的事，所以请做好准备花几个月甚至几年的时间去熟悉这个过程。

假设你需要设计一个断路器组件。断路器是一种设计模式，用于增强系统健壮性，保护系统不受其他组件（或者第三方系统）失败的影响。断路器在你的代码执行某个操作之前先去检查其可用性。代码清单 2-1 和代码清单 2-2 及图 2-6 和图 2-7 所示为断路器的设计。开始的时候先写主要接口（代码清单 2-1）的代码草稿，然后写使用者代码草稿（代码清单 2-2）验证接口。使用者代码可以是单元测试，也可以是一些无须编译的代码草稿。这个阶段仅仅是确认接口设计是明晰的易于使用的。然后你就可以画一个时序图展示使用者如何和断路器进行交互，以此进一步确认设计的正确性，如图 2-6 所示。最后，再画一个类图的草图，如图 2-7 所示，检查是否违反某些设计原则，结构是否简单清晰。

代码清单 2-1 正在设计的接口伪代码

```
interface Zend_CircuitBreaker_Interface
{
    public function isAvailable($serviceName);
    public function reportFailure($serviceName);
    public function reportSuccess($serviceName);
}
```

代码清单 2-2 客户端伪代码

```
$UserProfile = null;
if( $cb->isAvailable("UserProfileService") ){
    try{
        $UserProfile = $UserProfileService->loadProfileOrWhatever();
        $cb->reportSuccess("UserProfileService");
    }catch( UserProfileServiceConnectionException $e ){
        $cb->reportFailure("UserProfileService");
    }catch( Exception $e ){
        // service is available, but error occurred
    }
}
if( $profile === null ){
    // handle the error in some graceful way
    // display 'System maintenance, you can't login right now.' message
}
```

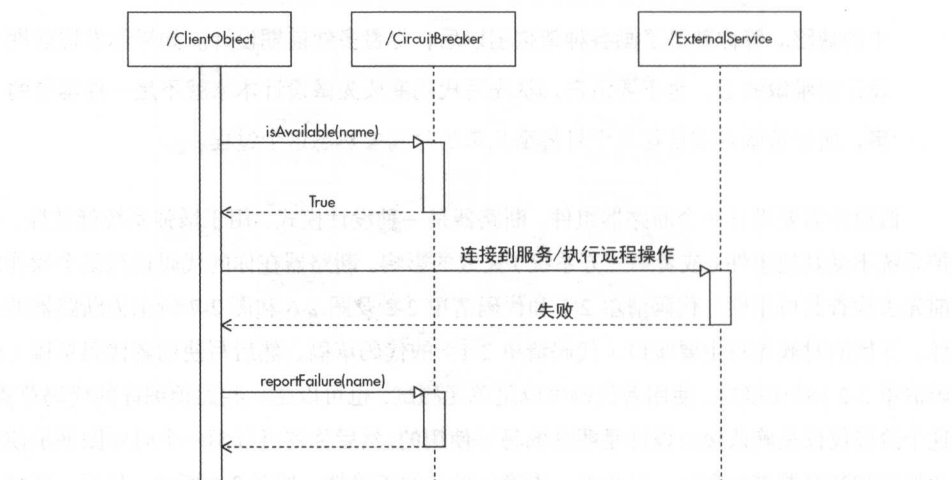


图 2-6 设计接口时，绘制序列图草图

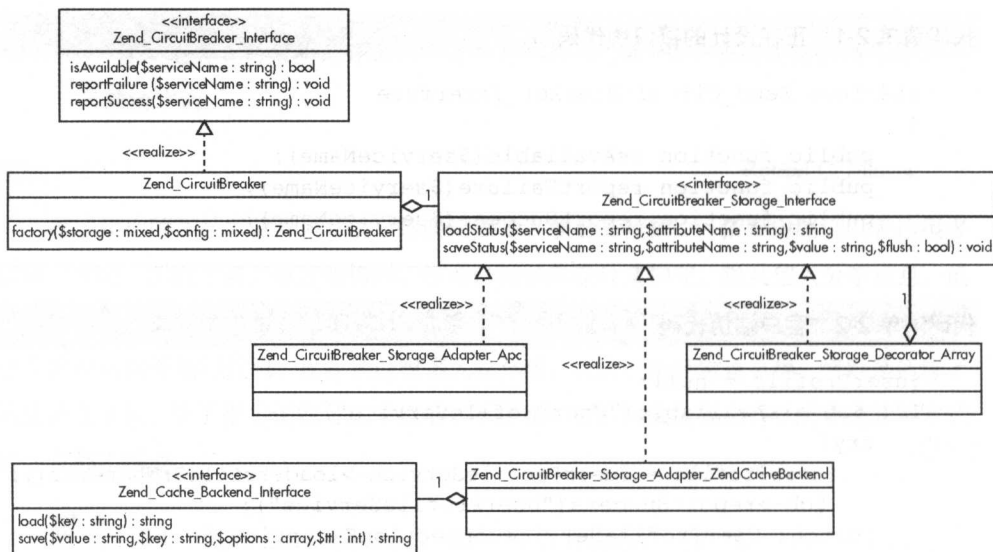


图 2-7 设计接口时，绘制类图草图

我相信遵循前面讨论的这些简单的设计过程就可以在创业公司做好前期设计工作了。你可以从多个角度审视自己的设计，并会因此获益良多。同时，你也可以降低开发风险，不至于做了一个不靠谱的设计，等发现的时候已经掉坑里了。

在写设计文档或者了解大规模系统的时候，有三种架构图特别有用：用例图、类图和

模块图。随着你的公司规模越大，你的团队人越多，你越会从这三种架构图中获益。让我们分别看看这三种架构图。

用例图

用例图是一种比较简单的图，定义了系统的用户是谁，他们可以执行哪些操作。用例图不考虑技术方案，仅仅关注业务需求，是一种将功能特性和业务需求提炼出来的好办法。当你要做一个新功能的时候，先画一个简单的用例图。用例图包含用一个小人图标表示的角色、角色可以执行的操作，以及各个操作之间的关系结构。用例图也可以呈现与外部系统的交互，比如远程 Web 服务 API 或者任务调度器。用例图不需要包含太多需求细节，做到简单、清晰、易于阅读就好了。画用例图的时候尽量在一个较高的层次上提炼关键操作和业务过程，而不是陷入到海量的细节中去。

图 2-8 所示为一个 ATM 应用的用户用例图例子。虽然 ATM 机包含大量的关于认证、安全、事务处理方面的细节，但是用例图只需要关心 ATM 完成用户的哪些操作就可以了。从这个角度看，按钮在屏幕上怎么排列或者 ATM 怎么去实现每个功能，这些都不重要。你只要关注一个需求的概要，就可以了解 ATM 系统的主要意图并定义各种约定。

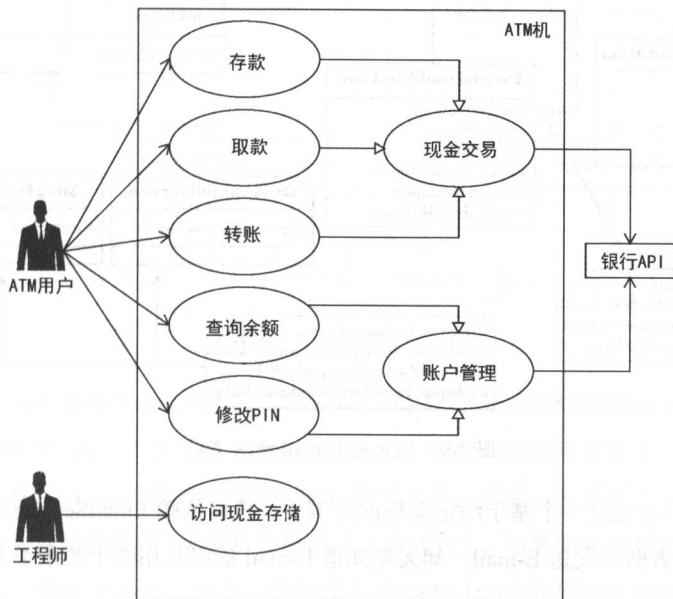


图 2-8 简化的 ATM 用例图

类图

类图展现独立模块的结构。一个典型的类图包含接口、类、关键方法名，以及它们的关系。由于每个类都用一个节点表示，每个依赖都用一条线表示，所以从类图上很容易看出耦合关系，可以看出哪些类耦合度更高，哪些类更独立。简单看每个节点上连接的线的数目就可以判断出这个节点包含的依赖有多少。类图是一种非常好的可视化手段，很好地展现了类、接口及其交互关系。

图 2-9 所示为一个类图的简单例子。这里的关键元素是类、接口、最重要的方法及依赖关系，不同的依赖关系用不同类型的线条表示。在这个例子中，EmailService 有两个实现类，一个使用 SMTP 协议直接发送 E-mail，另一个将 E-mail 加入队列延迟发送。

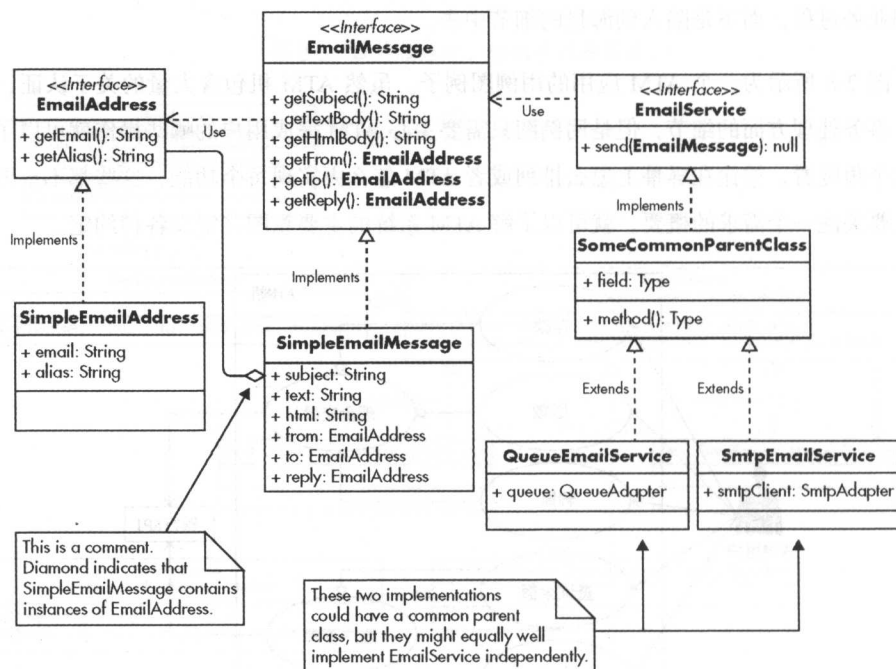


图 2-9 简化的 E-mail 模块类图

EmailService 也是一个基于约定编程的好例子，任何依赖 EmailService 的程序都可以使用 SMTP 或者队列发送 E-mail，却无须知道 E-mail 最终是用哪个实现类发送的。

接口只能依赖其他接口而不能依赖具体类，换个说法，要尽可能地依赖接口编程。

模块图

模块图很像类图，也是描述结构和依赖关系的。模块图和类图的不同在于模块图关注的抽象层次更高。模块图可以认为是代码层面的缩略图，是类图的上面一层架构图。模块图不再关注类和接口，而是关注系统更大粒度的组成部分，展示模块之间的依赖和交互。模块可以是一个包，也可以是逻辑上实现特定功能的组件。

图 2-10 所示为一个假想的支付服务（PaymentService）模块图的例子，展示了一个实现支付功能的应用中支付服务和其他部分之间的依赖关系。模块图通常关注应用中和特定功能有关的部分，而忽略那些不相干的部分。由于系统将来会变得更大，所以最好一开始就画多个模块图，每个模块图只关注一个特定的功能，而不是把所有的功能都画在一个模块图里。理想情况下，每个模块图都应该足够简单，你能全部记住并在脑子里重画一次。

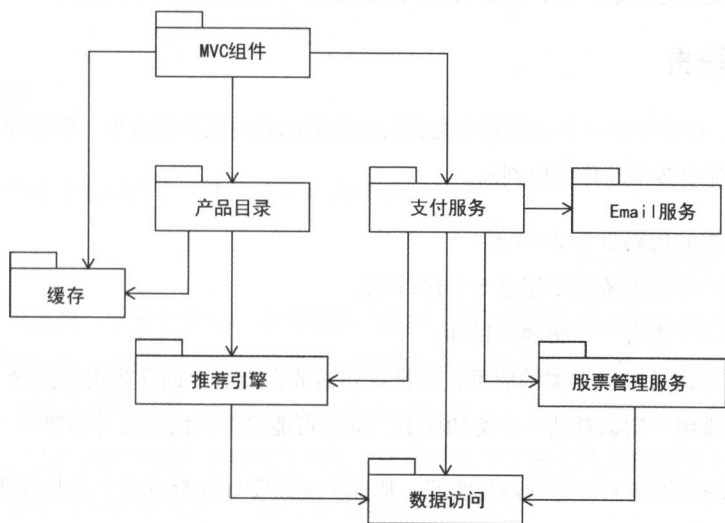


图 2-10 支付服务模块图

保持创新而不是担心自己画图的对不对。实践中，架构图容易理解比画得漂亮及符合规范标准更重要。进一步学习统一建模语言（UML）及设计模式可以参考一些书籍。^[1, 7, 10]另外，我推荐使用 ArgoUML 作为桌面 UML 绘图工具，这是一个开源的 Java 应用程序，可以在整个公司内协作使用，无须将软件设计上传到云上。如果你喜欢基于云的方案进行在线协作设计，试试 draw.io，这是一个集成了 Google Drive 的免费且容易使用的在线服务。Draw.io 是我的最爱，本书里几乎所有的架构图都是用 draw.io 画的。

单一职责

单一职责是降低代码复杂度的一个有效手段。单一职责原则是说你的类应该只有一个职责而不宜更多。单一职责可以降低耦合，增加简单性，易于重构，代码复用及单元测试——这些目前为止讨论过的所有的核心原则。遵循这个原则设计出来的类更简单更小，因此易于重构和复用。

有时候就眼前来讲，也许无视新功能是否为某个类的职责，直接往一个类里简单地添加方法更容易。然而，这样做几个月后，你的类会变得非常庞大和其他类的耦合也更紧密。你会看到类之间出现的各种交互操作并不像你期望的那样，这些类会进行一些无关的操作。同时，由于类变得很庞大，因此几乎很难完全理解它的行为和角色。随着时间的推移，这种情况会变得越来越严重，几乎每行代码都会带来复杂度的巨大增加。

改善单一职责

事实上，并不存在一个必须遵守的法则去衡量你的类是否符合单一职责原则。不过还是有一些最佳实践帮助你做出判断。

- 一个类的代码少于 2~4 屏。
- 确保一个类的依赖不超过 5 个接口/类。
- 确保一个类有一个明确的目标。
- 用一句话总结一个类的职责，并把这句话放在类名上面作为类的注释。如果你发现很难用一句话总结一个类的职责，那有可能这个类的职责并不单一。

如果你的类不能满足这些最佳实践，那你可能需要好好看看这个类并且准备重构。

在更高的抽象层面，你需要将功能按模块分隔并避免功能重叠。具体做法可以参照类的设计——试着用一句话总结模块或者应用的功能，不过要从更高层次总结。例如，你可以说：“文件存储系统允许用户上传文件、管理文件并支持复杂的文件搜索”，这样应用目标看起来就很清晰。可以使用一个明确的接口（比如一个 Web 服务定义）限制模块职责范围并将其从其他部分隔离开来。

单一职责的例子

我们看一个 E-mail 地址验证的例子。如果你把验证逻辑直接放在创建用户账号的代码里,就没办法在其他地方复用了。你就不得不复制粘贴这些代码或者在这些本来没什么关联的类之间进行一些让人难受的依赖,总之,你会破坏软件设计的核心原则。把验证逻辑剥离出来,你就可以保证其只有一个实现并能在其他地方复用。过了一段时间,你如果需要验证逻辑进行修改,比如需要在域名中支持 UTF-8 编码,只需要重构这个类就可以了。设计一个类专门负责 E-mail 验证比将这部分代码到处复制更简单,更容易将变化隔离开来。单一职责原则的另一个作用是你会变得更乐于写可测试的代码。因为类变得更少逻辑更少依赖,所以更容易进行独立测试。进一步掌握单一职责的办法是研究策略、迭代器、代理、适配器模式^[5,7]。这有助于你进一步了解领域驱动设计^[2]及更好的软件设计^[1,37]。

开闭原则

“好的架构就是可以让你延后做出决定”

—— Robert C. Martin

开闭原则是指,当需求变更或者增加新功能时,你不需要修改现有的代码。开闭的意思是:向扩展开放,向修改关闭。如果我们设计的代码在将来扩展新功能的时候无须修改,那么我们就遵循了开闭原则。正如 Robert C. Martin 所言,开闭原则可以让我们保留做出选择的余地,延迟到将来做出某些细节决定,这样就减少了现有代码的需求变更。这个原则的主要目的是增加软件的弹性,使将来变更的代价最小化。

用例子解释一下。考虑一个排序算法,你需要对一个记录雇员对象的数组基于姓名进行排序。一般实现中,你可以在一个类里包含算法及其他全部必要的代码,假设类就叫 EmployeeArraySorter,如图 2-11 所示。你可以在这个类里只暴露一个方法,允许对雇员对象数组进行排序,然后就可以说这个功能开发完了。虽然这样是能解决问题,但是显然不够有弹性,事实上,这是一个非常僵硬的实现。由于所有的代码都在一个类里,你几乎不能在不修改代码的情况下扩展或者增加任何功能。假如你有一个新的需求,需要对城市基于人口进行排序,就不能复用这些已经开发好的代码。这时,你就不得不面临一个尴尬

的选择，要么扩展 `EmployeeArraySorter` 做一些和原先设计完全无关的事，要么复制粘贴这个类的代码然后再修改。幸运的是，你还有第 3 个选择，就是重构这个类使其符合开闭原则。

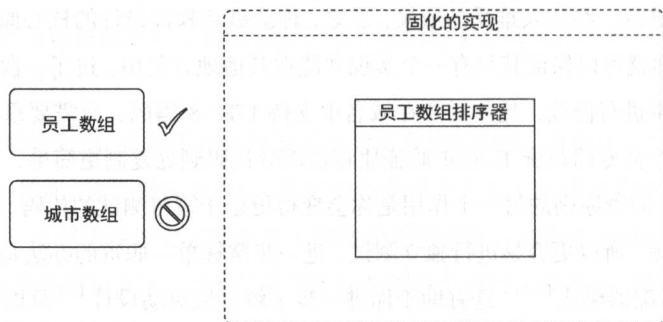


图 2-11 固化的实现

开闭原则需要你将这个排序问题分解成几个更小的任务。每个任务都应该是独立的，不影响其余部分的复用。你可以设计一个接口 `Comparator` 用来比较两个对象，再设计一个接口 `Sorter` 用来执行排序算法。`Sorter` 会利用 `Comparator` 对输入的数组进行排序。设计如图 2-12 所示，这个设计有点类似 Java `Comparator` API 的设计。

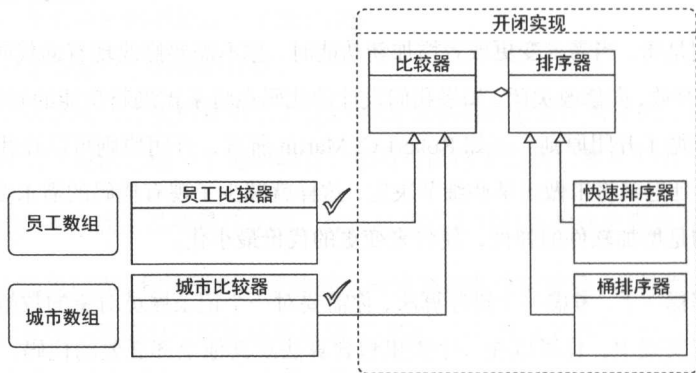


图 2-12 开闭实现

使用这样的设计可以很容易地复用现有的代码。例如，你想改变排序对象，只需要增加一个新的 `Comparator` 实现即可，无须对现有代码做任何修改。如果你想改变排序算法，也不需要修改 `Comparator` 或者 `Sorter`，只要重新创建一个 `Sorter` 接口的实现即可。通过让

各个不同部分独立变化，可以减少变化影响的范围，也可以扩展现有的代码而无须对它们做任何改动。

另一个开闭原则的好例子是各种 MVC 框架。这些框架由于其简单和易扩展的特性而广泛应用于 Web 开发中。回想一下，你什么时候需要修改 MVC 框架的某个组件？如果这个 MVC 框架的架构设计得足够好，答案是永不需要。不过，虽然不需要修改，但你还是可以扩展 MVC 中的某个组件，增加新的路由、拦截请求、返回不同的响应、覆盖缺省的行为。你不必修改现有的框架代码，就能扩展其功能，这就是开闭原则所起的作用。

和掌握其他设计原则一样，学习开闭原则，开始的时候先熟悉了解各种框架。观察开闭原则的各种实现思路，领会开闭原则的实现模式。例如，Java 语言实现的 Spring MVC 框架就是一个很好的开闭原则的例子。用户可以弹性地实现各种功能而无须修改框架本身，而客户端代码跟框架也几乎没有多少耦合。通过使用 annotation 和基于约定编程，你开发的绝大部分类甚至都不知道 Spring 框架的存在！

依赖注入

我们已经在本章的前面探讨过依赖的问题，这是关于耦合和复杂性最重要的话题之一。依赖注入是一种降低耦合改善开闭特性的简单技术。依赖注入对类需要依赖的对象提供一个引用实例，而无须类自己创建依赖对象。依赖注入的核心思想是知道得越少越好。尽可能地让类不要知道如何去创建需要的依赖，以及这些依赖来自哪里，是如何实现的。这看起来只是从“拉取”变成“推送”的一个微小变化，但事实上会对软件设计的弹性产生巨大影响。

我们来看一个 CD 播放器的依赖注入的例子。所有的 CD 播放器都持有一个 CD 接口，知道如何去读取音轨，如何对音乐进行解码，读取 CD 内容的雷射器的光学参数是什么。插入到 CD 播放器的 CD 是一个依赖，没有 CD，CD 播放器就没法正常工作。将发现依赖的职责交给用户，CD 播放器就可以保持简单。从而使 CD 播放器的复用性更强，它不必知道烧录到 CD 上的每一个标题，也不必知道专辑上的每个组合，只需要知道 CD 的各种可能形式及其组合形式就可以了。CD 播放器需要你（用户）去提供一个可读的 CD，一旦你提供了一个 CD 实例，CD 播放器就可以工作了。

此外还有一个附带的好处就是 CD 播放器允许插入非标准的 CD。你可以插入一张空白盘，或者一张音轨畸变的测试盘，这样你就可以做异常测试了。

图 2-13 所示为一个负担过重的 CD 播放器，它通过硬编码的方式记录各种 CD，如果你想播放一张新 CD，你就不得不修改它的代码。

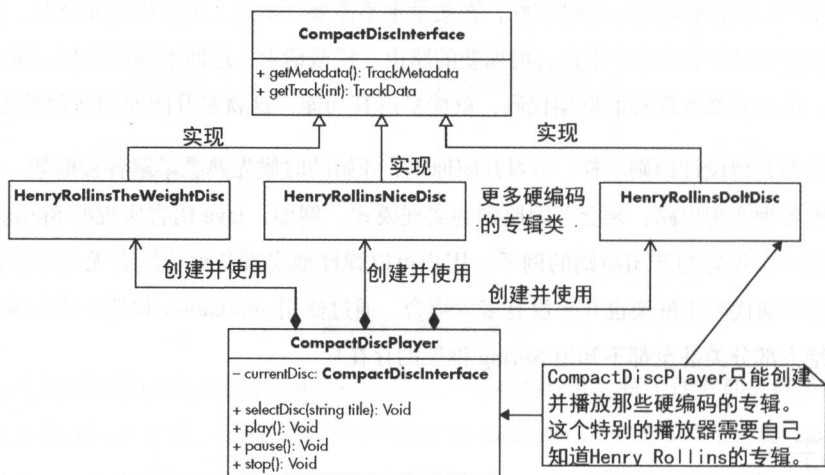


图 2-13 没有依赖注入的 CD 播放器

我们再看一下使用依赖注入方式实现的 CD 播放器，如图 2-14 所示。它不必知道任何 CD 自身的信息，只需要提供一个 CD 的实例就可以。使用这种方式，你可以实现对新开放（使用新的 CD）而对修改关闭（CD 播放器永不需要修改）。

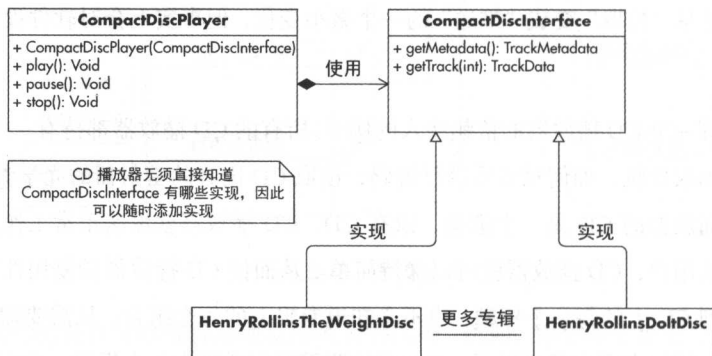


图 2-14 使用依赖注入的 CD 播放器

实践中，依赖注入通常认为类中不要使用 `new` 这个关键字，需要依赖的对象实例由调用者提供。代码清单 2-3 所示为一个基于构造函数的依赖注入的 Java 例子。使用这种方式，实例创建时就拥有了全部功能，CD 播放器所能做的事情都是可预期的。把收集的各种依赖的职责从类中剥离出来，使得类更简单，更可复用，更可测试。

代码清单 2-3 基于构造器的依赖注入的例子

```
class CompactDiscPlayer {
    private CompactDisc cd;
    public function CompactDiscPlayer(CompactDisc cd) {
        this.cd = cd;
    }
    // other methods and business logic
}
```

如果使用得当，依赖注入可以有效降低类的局部复杂度，使其更简单。无须知道谁来提供依赖的实例，也无须知道实例从何而来，这样类就可以聚焦自己的主要职责。类的代码更简单，不需要太了解系统的其他部分，不太需要变化和单元测试。将集成的代码从类中移除，可以使类更加独立、可复用、可测试。依赖注入在面向对象编程（OOP）社区已经有多年实践。实现依赖注入不需要任何框架。学习依赖注入我推荐 Spring 框架或者 Grails 框架，它们是依赖注入方面非常好的例子。推荐阅读文献^[w76, 1, 14, 22, 71]。

控制反转（IOC）

依赖注入是非常重要的一个原则，也是范围更广的控制反转原则的一个子集。依赖反转主要解决对象的创建及依赖的组合，而控制反转则更通用，可以在各个抽象层面解决各种问题。

控制反转（IOC）是一种从类中移除职责的方法，从而使类更简单，与系统其他部分更少耦合。控制反转的核心是不必知道是谁、怎样、何时创建和使用对象，尽可能地简单易于理解，对于软件设计而言，各部分互相知道得越少越好。

IOC 在一些框架中重度使用,包括 Spring、Symfony、Rails,甚至 Java EE 容器。你不必创建类调用方法,这些统统交给框架来做。IOC 框架接收 Web 请求创建对应处理类的实例交给对应的模块处理。

IOC 也被称为好莱坞原则,好莱坞原则是说“不要给我们打电话,我们会给你打电话”。运用到实践中,就是你的类实例不必知道自己什么时候被创建,被谁使用,自己的依赖又是如何被创建的。你的类是一个插件,外部力量决定这些类什么时候被创建,如何被使用。

想象一下只用纯 Java 不需要任何框架开发一整个 Web 应用。没有 Web 服务器,没有框架,没有 API,只有 Java。为了完成这样一个复杂的应用,你需要自己写非常多的代码。即使你决定使用一些第三方的库,你也必须自己控制整个应用的流程。通过使用框架,你可以减少代码的复杂度。不止是可以减少需要写的代码的数量,还可以减少开发者需要关注的事情。需要做的只是让框架回调你的代码,框架会创建类的实例。请求到达的时候,框架会调用你的方法进行处理,并控制执行的流程从一个扩展点到另一个扩展点。

图 2-15 所示为一个用纯 Java (没有用任何框架)编写的 Web 应用。这个例子中,大量的代码模块用于和外界进行通信。应用需要自己打开网络端口,写日志文件,连接外部系统,管理线程,解析消息。应用需要控制所有的事情,这意味着你需要考虑太多与业务无关的事。

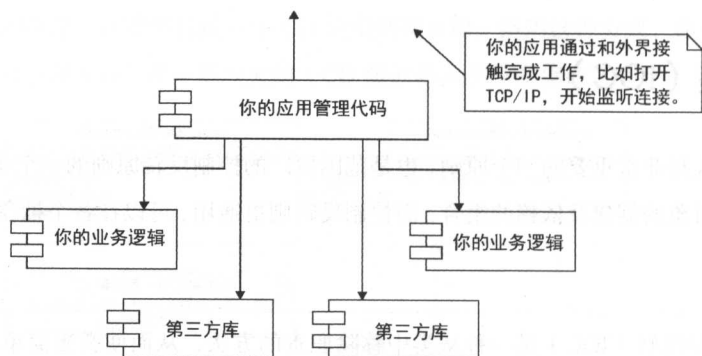


图 2-15 用纯 Java 写 Web 应用,没有 IOC 框架

如果你使用 IOC 框架,应用看起来如图 2-16 所示。框架处理了大量的非业务事务,应用甚至不需要关心这些事情的存在。虽然系统还是要处理这些事情,但是都和应用无关。这不会改变整个系统的复杂性,但是会极大地降低应用的复杂性。

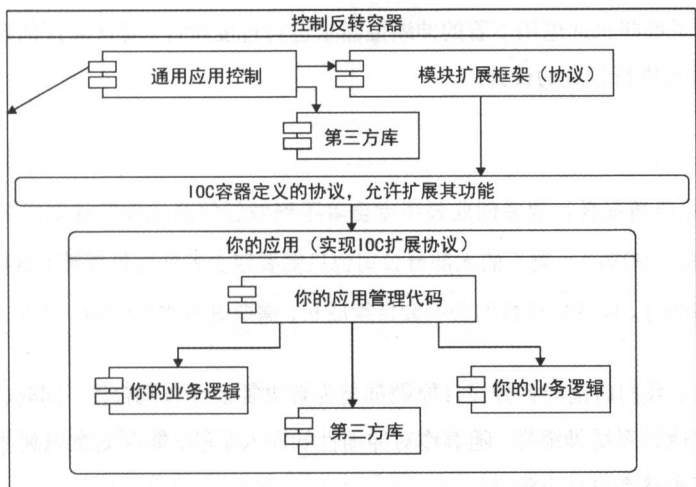


图 2-16 使用 IOC 容器的 Web 应用

IOC 是一个很通用的概念。你可以为任何类型的应用创建各种控制反转的框架，不只是 MVC 框架或者 Web 请求处理框架。一个好的 IOC 框架包含如下特性：

- 可以为框架开发插件。
- 插件是独立的，可以被随时插入移除。
- 框架可以自动检测到插件，或者通过配置文件加载插件。
- 框架为每一种插件类型定义接口，而不会与具体插件耦合。

提示

基于 IOC 框架写代码就像把鱼放在鱼缸里。鱼缸里可以放很多鱼，它们互不干扰，但是它们生活在一个它们不能控制的世界里。你决定鱼缸的环境是什么样的，什么时候喂鱼。你就是一个 IOC 框架，你的鱼就是插件，它们生活在一个被保护的自己却不知道的泡泡里。

为伸缩而设计

为伸缩而设计是一个有难度的艺术活，本节中描述的每一项技术都要付出相应的成本。作为一名工程师，你需要在没完没了的伸缩和每一种对应的方案实践之间进行仔细的

权衡。不要为了那些永远也用不着的伸缩性需求进行过度设计，要认真评估系统最可能的实际伸缩性需求进行相应的设计。

提示

从业界情况看，很多创业公司完全等不到做任何系统伸缩就倒闭了（这样的公司大概占 90%）。剩下的大部分公司也只需要较少的伸缩性就够（这样的公司占了大概 9%）。只有极少数的公司会持续成长，需要进行水平伸缩（只占 1% 左右）。

目前为止，我们讨论的各种设计原则都是为解决复杂性和耦合性，同时，这些原则大部分也有助于改善系统伸缩性。随着你对伸缩性的深入了解，能深刻意识到伸缩性方案都可以浓缩成三个基本设计方法。

- **增加副本**：同一组件重复部署到多台机器。
- **功能分割**：基于功能将系统分割成更小的子系统。
- **数据分片**：在每台机器上都只部署一部分数据。

这些方法会提供某一方面的好处，同时也需要付出另一方面的成本。需要对这些方法非常熟悉，才能将它们灵活应用到系统设计中。我们用一个例子更深入地了解这些方法。设想我们开发一个 Web 应用让人们可以管理他们的 eBay 拍卖标的。用户创建账号，让应用代替他们去竞标，有趣又简单。

增加副本

如果你从头开始构建一个系统，最简单最常见的伸缩性策略是增加副本。副本指的是组件或者服务器的副本。任何时候两个副本都是可以互换的，任何请求在任何副本上处理都是一样的。换句话说，发送请求到任何一个副本上得到的响应都是一样的。

对于我们这个 eBay 竞价的应用，如果应用受欢迎，就需要对应用的所有组件进行伸缩。如我们第 1 章提到的，你可以对现有服务器进行升级（垂直伸缩），也可以增加服务器分担负载（水平伸缩）。增加副本进行伸缩的方式更适合 Web 应用，所以我们优先考虑。图 2-17 所示为将 eBay 竞价应用部署在单一服务器上的情况。

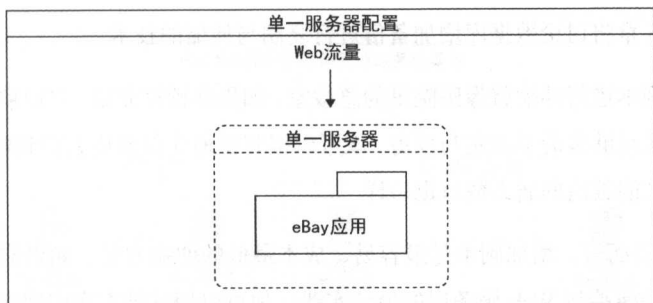


图 2-17 单一服务器配置

通过增加副本实现伸缩性，构建一组可交换的 Web 服务器平均地分担整体的负载压力。在这种部署模式下，系统负载（Web 请求）会通过负载均衡分发到多台副本服务器上。理想情况下，负载均衡收到 Web 请求的时候，可以将请求分发到任何一台副本服务器上而无须知道前面请求的分发情况。图 2-18 所示为同一个应用通过增加副本实现伸缩。

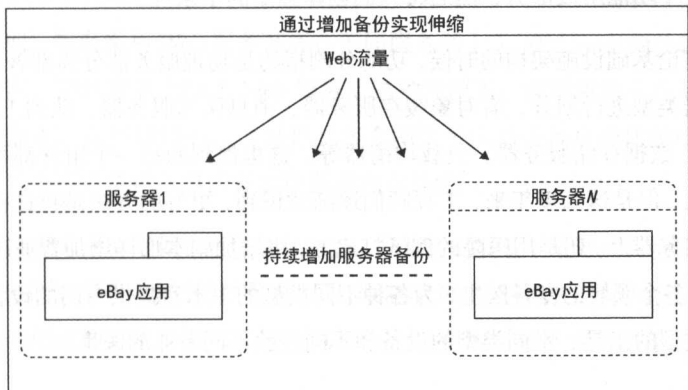


图 2-18 增加备份实现伸缩

通过增加副本实现伸缩时，需要关注应用状态如何存储及副本之间如何同步状态变更。所以这种方式最适用于无状态的服务，无须同步状态。如果应用是无状态的，那么对于请求而言，一台新服务器和一台已经存在的服务器没有区别。在这个例子中，你可以简单地增加服务器到负载均衡池中实现处理能力的提升。（无状态服务是指一个服务不会依赖本地状态，处理一个请求也不会影响到服务自身的行为方式。要获得正确的结果无须特定的服务器实例。更多无状态服务的细节将会在第 3 章和第 4 章讨论。）要注意的是，增加副本实现伸缩不是无状态服务的专利。事实上，数据库使用这种方式通过副本进行伸缩

已经多年。第 5 章将讨论数据库增加备份进行复制与伸缩的技术。

通过增加副本进行伸缩就像医院里的急救室。如果你预算充足，可以雇用足够多的职业医生并且购买足够多的手术室与设备，就可以很容易地让很多病人得到救治。医生和病房的数量一样，能救治的病人数目也一样。

对于 Web 层而言，增加副本是最容易、成本最低的伸缩方案。如果你遵循第 3 章和第 4 章将描述的前端与 Web 服务层的最佳实践，你可以用这种方案实现应用的伸缩性。增加副本实现伸缩的主要挑战是对于有状态的服务难以用这种方式伸缩，需要找个办法同步状态以实现副本任意可交换。

功能分割

第二个重要的伸缩性策略是功能分割，适用于各个地方各个层面。这个方案的核心是发现系统的各个功能组成部分，然后将它们创建独立的子系统。

当我们讨论基础设施架构的时候，功能分割指的是物理服务器分离部署。将数据中心的服务器根据类型进行划分，有对象缓存服务器、消息队列服务器、队列工作者服务器、Web 服务器、数据存储服务器、负载均衡器等。这里提到的每一个组件都可以部署到主应用服务器上，但是这么多年来，工程师们逐渐意识到，更好的办法是将这些功能组件部署到独立的服务器上。还是用医院的例子打比方，跟增加副本那样增加普通医生的做法不同，可以雇用各个领域的专科医生，为各种不同类型的手术室提供专门的设备。不同的急诊需要不同类型的工具、不同类型的设备和不同经验不同专业的医生。

功能分割的另一种更先进的方式是：将系统切分成一些自给自足的应用。主要适用于 Web 服务层，是面向服务架构（SOA）的一种主要实践。回到 eBay 竞价应用这个例子，如果你有一个 Web 服务层，就可以构建一系列的松耦合的 Web 服务处理实现各种功能。这些服务可以拥有自己的逻辑资源，诸如数据存储、队列、缓存等。图 2-19 所示为一个功能分割的场景，两个独立的功能：资料服务和调度服务。这些服务可以共享底层基础设施，也可以拥有独立部署的底层基础设施，比如数据存储等。给服务更多的自主性，一方面可以更好地基于约定编程，另一方面也可以让服务自己决定需要什么样的组件及如何对自己进行伸缩。

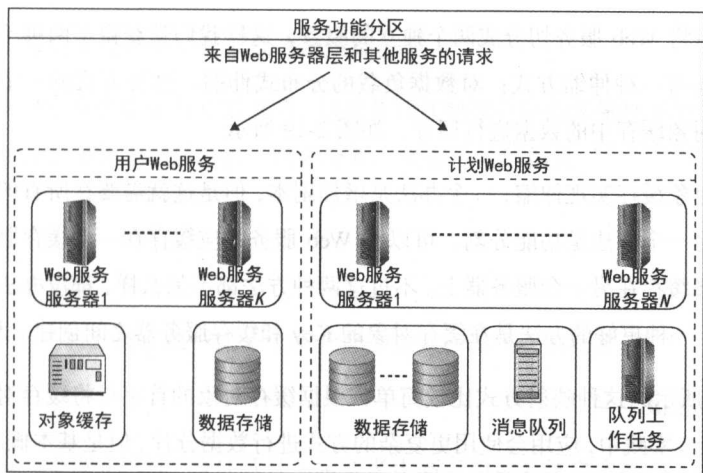


图 2-19 功能分区

功能分割常用于底层：将应用分割成多个模块，然后将不同类型的软件部署在不同的服务器上（比如，数据库和 Web 服务部署在不同服务器上）。在大一点的公司，也会对顶层进行功能分割，即创建多个独立 Web 服务。这种情况下，需要将单个应用切分成多个小一点的功能服务。这样做还有一个好处是多个团队可以基于各自的代码库并行独立开发，不同的服务也可以根据各自的伸缩性需求独立伸缩。

功能分割会带来很多好处，但是也有缺点。分割后的功能，最开始时会带来很多管理方面的困难。功能分割可以分割的数量有上限，限制了对功能分割这种伸缩性方案的使用。你不能无限地使用这种方式对应用无休止地切分，以使 Web 服务变得更小。

数据分片

第三种主要的伸缩性策略是将数据分片，然后将不同的分片数据保存在不同的机器上，以保证每台服务器上存储的数据量都不会太大。这种方式也被称为无共享架构原则，每台服务器都拥有自己的一个数据子集，彼此之间完全独立，每个节点都完全自治。这样，每个节点都可以独立变更自己控制的部分数据，而无须在各个节点之间复制状态变更信息。无共享状态意味着没有数据需要同步，不需要锁，失败也会被隔离，因为节点之间没有依赖。

我们还是来看 eBay 竞价应用。回顾一下，首先我们通过增加服务器（增加备份）实

现伸缩，然后将 Web 服务切分成两个独立的服务，这样我们就获得了两种不同的伸缩方式。此外，还有一种伸缩方式：对数据负载的分布式伸缩。这种方式的一个例子是：对 Web 服务的对象缓存中的数据进行切分，如图 2-19 所示。

要想让对象缓存实现伸缩，一个办法是增加副本，但是这就需要在所有的服务器之间同步状态。另一个办法是功能分割，可以将 Web 服务响应缓存在一台缓存服务器，而数据库查询结果缓存在另一台服务器上。不过这两种方法都不怎么样，都没办法让我们的伸缩持续下去。一种更好的办法是在缓存对象的 Key 和缓存服务器之间创建一种映射关系。

图 2-20 所示的这种映射方式比较简单，根据缓存对象的首字母将缓存对象分布到不同的服务器上。实践中，应用会使用更复杂的方式进行数据分片，但是基本概念是一样的。每个服务器都得到一个数据子集，并只需管理这个子集即可。由于每个服务器都只存储较少的数据，所以服务器可以把更多数据存储在内存中并得到更快的响应速度。将来如果需要增加更多存储能力，只需要增加服务器并修改映射关系即可。

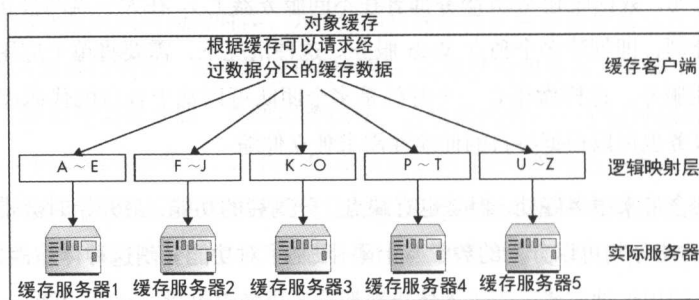


图 2-20 数据分区

还是用医院做类比。如果医院提供预约就诊服务，就可以用数据分片的方式对专家进行伸缩。不能让病人每次预约到的都是不同的专家，解决这个问题一个办法是将医生的名字写到每个病人的病例卡中。病人来挂号时就把他分配给对应的医生，这样客服人员可以很容易地根据预约帮病人找到医生。

数据分片配合增加备份，可以实现几乎无限的伸缩性。只要能对数据进行正确的切分，就能增加更多的用户，处理更多的并发连接，收集更多的数据，将系统部署在更多的服务器上。不幸的是，数据分片几乎是最复杂、代价最昂贵的技术。数据分片最大的挑战是在进行数据访问前，必须先找到存储需要的数据所在分区的服务器，而如果一个查询涉及多

个数据分区，那么实现就变得异常低效和困难。

第5章将讨论更多数据分片的细节，这也是现代数据存储的关键性伸缩技术之一。你也可以自己阅读更多关于增加备份、功能分割、数据分片的信息。^[27,41,w34,w35]

自愈设计

“一个足够大的系统一定处于某种部分失效的连续状态之中。”

——Justin Sheehy

本章最后一个设计原则：设计软件一定要考虑高可用和系统自愈能力。一个系统被认为是可用的，就是说从用户角度它一直按预期运行，完成其功能，而并不关心其内部是否存在某种失效状态，只要不影响用户使用就可以。换句话说，你需要让你的系统表现出所有组件都在正常运行，即使出现某些设备宕机或者进行发布维护也是如此。

一个高可用的系统对于其用户而言，任何时候都是可用的。对于高可用没有一个绝对的衡量准则，不同的系统有不同的高可用需求。系统的高可用性一般不会用一个绝对值直接度量，而是用“几个9”来衡量。我们说一个系统两个9，是说这个系统99%的时间可用，也就是每年大概有3.5天不可用（ $365 \text{ 天} \times 0.01 = 3.65 \text{ 天}$ ）。相对地，一个系统有5个9的可用性，是说这个系统99.999%的时间可用，每年不可用时间5分钟。

可以想象，不同的业务场景有不同的可用性要求。要记住的是，系统越大，系统失效的概率越高。如果你需要连接5个Web服务，每个服务连接3个数据存储，那么你就需要依赖15个组件。任何一个组件失效，整个系统都会不可用，除非你能优雅地控制失效进行透明地失效转移。

当你进行伸缩扩容的时候，失效也会变得更加频繁。你有1000台服务器，那么每天都可能有几台服务器宕掉。糟糕的事情可能远不止于此，很多原因都会导致失效，比如停电、断网，以及人为失误。设计一个伸缩性架构必须把各种失效状态当作常态，而不是特殊情况对待。如果想设计一个高可用的系统，就必须做最坏的准备才能得到最好的期待，

要不停地想哪里会出错，出错后该怎么办。

保证系统高可用是一种态度，将这种态度发挥到极致的一个例子是 Netflix 开发的一个叫做 Chaos Monkey（捣蛋鬼）的系统。Netflix 可用性工程师认为证明一个系统真的高可用的方式是，真的去制造一些事故，然后看这个系统怎么处理。Chaos Monkey 就是这样一种服务，它会在上班时随机关掉一些 Netflix 的基础设施组件。这看起来很荒谬是不是？公司可能都会因此而挂掉。但是最后真的证明了他们的系统可以处理任何类型的失效。

另一种类似的高可用态度是一个叫做 Crash-Only（只需宕机）的概念。Crash-Only 的鼓吹者认为系统应该随时为宕机做好准备，系统任何时候重启，都无须人工干预。这意味着系统无论是否正在处理请求，处理队列消息还是做任何类型的工作，都能够自己检测到自己的失败，修复必要的错误数据、重启，然后像正常时一样工作。CouchDB 是一个很流行的开源数据库，它就是遵循这个原则开发的，这个系统不提供任何的关闭功能。如果想关闭一个 CouchDB 实例，只需要终止进程即可。

如何才能证明你的系统能够处理各种失效状况？我曾经见证过很多宕机事件，有的因为本地服务器存储了状态，有的因为不能正确处理网络延迟。持续地测试各种失效场景是提高系统可用性的重要方式。实践中，确保系统高可用的主要手段是消除失效单点的风险和优雅地进行失效转移。

失效单点是指基础设施中任何一个部分对系统的正常工作都是必要的。失效单点的一个可能的场景是域名系统（DNS）服务器，如果你只有一个域名服务器的话。另一个可能的场景是数据库主服务器或者文件存储服务器。

识别失效单点的一个简单方法是画出数据中心架构图，每个设备（路由器、服务器、交换机等）都画上去，然后问自己，当某个设备宕机的时候会发生什么。当识别出失效单点以后，就和业务团队一起讨论对其做冗余是否划得来。有时，冗余很便宜很简单，有时又很昂贵很复杂，这需要仔细权衡。

冗余是指数据或者组件有至少两个备份。当其中一个备份失效了，系统可以使用另一个备份服务用户。一个缺乏冗余的系统需要特别关注，这方面的最佳实践是准备一个灾难恢复计划（有时候也叫业务持续计划），记录对系统各种灾难性问题如何进行恢复。

最后,如果你想要一个高可用的完全耐受各种失效的系统,也许该实现系统的自愈功能。自愈比优雅地处理失效更进一步,它能够检测并自动修复问题,无须人工干预。能自愈的系统是 Web 应用的圣杯,不过想打造这样一个系统也是非常困难且代价高昂的,比嘴上说说不知道要难到哪里去。这里给一个自愈的例子——开源数据库 Cassandra 处理失效的办法。Cassandra 中系统可以透明地处理数据节点的失效。一旦集群发现某个节点失效,就立刻停止发送新的请求给失效的节点。对于客户端而言,失效的时间就是发现失效节点需要的时间。一旦失效节点被放到黑名单里,客户端就可以正常读写数据,集群中其余的节点会对失效的节点提供数据冗余。当失效节点恢复上线后,会带着丢失的数据运行,好像系统什么都没发生一样。

相同地,如果用一个新的空白服务器节点替换一个失效的节点,也不需要系统管理员像传统关系数据库那样,必须从备份节点恢复数据。增加一个新的空白数据节点会引起 Cassandra 集群的数据同步,过一段时间,新加入的机器就填满了数据。如果一个系统能够检测到自身存在部分失效或者潜在的不可用,并且能尽快进行自我修复,这个系统就是自愈的。系统能在最短时间恢复,并且能够自动化地完成这个过程,这就是关于自愈的一切。

平均恢复时间是可用性方程的一个重要组成部分。你越快检测到,去处理,进行修复,可用性就越高。可用性的真实计算公式是: $\text{平均失效时间} / (\text{平均失效时间} + \text{平均恢复时间})$ 。减少平均恢复时间,就可以增加可用性,即使失效次数没法控制的情况下也是如此。当使用亚马逊 Web 服务(AWS)这类云主机服务时,由于云服务商使用廉价硬件,他们会在低失效比率和低价之间权衡。这种情况下,你没法控制平均失效时间,所以只能更多地关注平均恢复时间。

我强烈推荐读者学习更多关于高可用、监控、自愈系统方面的知识。这方面也有很多参考资料^[w35, w4, w7, w1, w15, w18, w27, w36, w3, w42]。第 5 章将讨论各种存储引擎的高可用。

小结

无论你是软件工程师、架构师、开发组长、还是技术经理,软件设计原则对你都非常重要。软件工程是一门关于信息决策、创造商业价值、为未来做准备的技术。要记住:设

计原则是你的指南针，是你的北斗星。它们能为你指明方向，增加你成功的概率，但是最终，还是要你自己决定什么样的方式最适合你的系统。

作为一名软件工程师或者架构师，你的工作就是根据金钱、时间、能力，提出对实现商业目标最好的解决方案。如果你意识到自己角色的重要性，你就应该让自己的思想开放，用各种视角考虑问题。最“干净”的方案不一定总是最好的方案，因为可能会花费更多的开发时间或者会引入一些不必要的管理成本。比如，解耦合和过度设计之间的那条线就非常细。你的工作就是识别出这些诱惑，避免带着美好的想象向着那些似乎酷毙了的方向越走越远。根据业务需求做出决策，在伸缩性、弹性、高可用、成本、推向市场的时间之间做出权衡。本书最后一章将探讨针对各种常见情况如何进行权衡。

要实事求是。如果你确信对你的业务有利，对你的软件有利，就不要怕打破这些规则。每个系统都是不同的，每家公司的需求也是不同的，你要发现自己和以前和别的工程师工作场景的不同之处。构建可伸缩软件的路不止一条，技术要好，开发工具要顺手，还要去发现业务驱动因素。希望本章提到的各种设计原则为你设计高质量软件系统开了个好头。加油！

3

构建前端层

前端层横跨多个组件，包括客户端（通常是 Web 浏览器）、客户端和数据中心之间的网络，以及数据中心响应用户请求的部分。前端是系统防御的第一线，主要的访问流量都会经过这里。用户的每次交互，每个连接，每个响应都需要穿越前端各个组件。这就需要前端具有很大的吞吐能力，并发处理能力，也就是前端要极具扩展性。幸运的是，如果设计良好，前端的扩展性是很容易做到的，此外前端缓存的收益也是极高的。

理想情况下，前端应用几乎是无状态的，可以简单地通过增加硬件实现水平伸缩；同时前端也非常依赖缓存。

在我们进行更深入的学习之前，先了解构建前端应用的各种手段。目前主要的 Web 网站构建方法有：传统的多页 Web 应用、单页 Web 应用（SPA），或者这两者的混合应用。

• 传统多页 Web 应用

每次点击按钮或者链接时，这类 Web 应用就会发起一个新的 Web 请求，浏览器从服务器收到响应后，会重新加载整个页面。当万维网刚刚出现时，使用的就是这种模式，当时没有 JavaScript，没有 AJAX，没有 HTML5。尽管现在 20 年过去了，但是依然可以使用这种方式构建可伸缩的 Web 网站（主要是因为其简单）。

• SPA

不同于传统或者混合应用，这类 Web 应用在浏览器执行大多数业务逻辑。这类应

用主要使用 JavaScript 开发，Web 服务器退化成一个数据应用程序接口（API）和一个安全层。这种模式下，任何时候你在用户接口上执行一个操作（比如点击一个链接或者输入一段文本），JavaScript 代码都会异步调用服务器加载/保存数据。SPA 模式最近几年变得越来越流行，出现了许多很不错的框架，诸如 AngularJS，以及移动 APP 上的 Sencha Touch 和 Ionic，但是总体上还是不如混合模式流行。SPA 模式的主要好处是可以提供更丰富的用户接口，同时，也会有效降低网络通信开销，缩短用户交互的延迟。

● 混合应用

这种方式是目前最主要的 Web 应用开发模式。正如字面意思，这类应用是传统多页 Web 应用和 SPA 的混合。部分交互是整个页面加载，部分交互使用 AJAX 实现局部页面更新。使用 AJAX 的同时又采用多页结构的方式使开发工作更具灵活性。在开发富用户接口的同时对搜索引擎优化（SEO）也更友好，开发相对更简单。

本章主要讲述的表示层组件就是这三种模式，主要关注的是混合模式和传统模式。如果你打算开发一个纯的 SPA 模式应用，开发模式和缓存需求都会很不一样，这超出了本书的讨论范围。

状态管理

“高效利用资源的关键是无状态自治的计算节点。”

—— Bill Wilder

状态管理是 Web 前端伸缩性最重要的方面。如果你能从前端服务器中移除全部的状态，就能简单地增加服务器实现前端伸缩。我们先看一下有状态服务和无状态服务之间的不同，然后简单探讨如何处理这两种不同类型的状态。

无状态是服务、服务器或者对象的一个属性，意思是它们不持有任何数据（状态）。无状态可以做到相同目标实例任意交换，因此具有更好的伸缩性。由于不持有任何数据，从客户端视角看，任何服务实例都是相同的。无状态服务自己不持有数据，它们将状态管理委托给其他的外部服务。

图 3-1 所示为一个有状态 Web 应用服务器的抽象架构图。服务器实例 A 持有其他实例 (B 和 C) 不能访问的信息: 用户会话信息、本地文件、本地内存数据, 甚至是锁。当我们讨论有状态和无状态的时候, 状态可以是任何信息, 需要在不同服务器之间进行同步, 以使它们对外看起来完全一致。我们再看一下图 3-2, 了解无状态服务器是如何处理用户状态的。在这个例子中, 服务器 A、B、C 是一样的, 所有的状态都保存在外部。这些服务器都是可交换的, 因为它们对用户请求的处理结果都一样。



图 3-1 有状态的服务器

为了更好地理解有状态和无状态的区别, 我们用去酒吧喝酒打个比方。如果你去的是一个很大的酒吧, 通常会看到很多吧台, 它们坐落在酒吧的各个楼层各个角落。在这里, 酒吧就是一个 Web 网站, 吧台就是服务器, 点一杯酒就是一个 Web 请求。

如果你用现金支付, 你可以径直走到任意一个吧台, 点一杯酒, 付款, 然后就可以端着酒走开。整个交易过程无须其他额外的步骤。服务员无须知道你的名字, 你可以找任何一个吧台的任何一个服务员买酒喝。从服务员的角度看, 他不在乎同时在酒吧的人有多少, 这不会影响到他卖酒。他可能会收到很多买酒请求, 但是这些请求之间互相不会影响。这是无状态服务的工作模式。

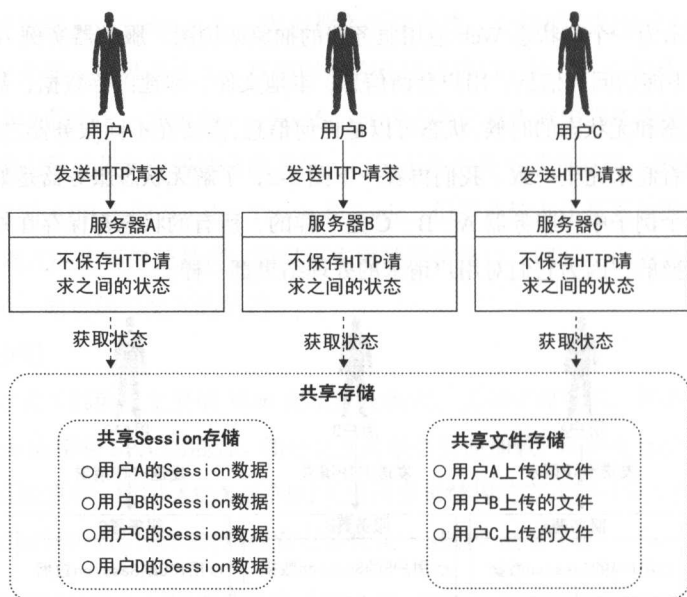


图 3-2 无状态的服务器

如果你决定用签单的方式买酒喝，整个交易就完全不一样了。首先，你需要给服务员你的信用卡以便建立一个账单。当你想要喝一杯的时候，你必须要到建立账单的吧台，买一杯酒并把账记在账单上。不能自由地在酒吧游荡选择其他的吧台，他们并不知道你是谁。醉酒而归的时候，你还要记得去吧台要回你的信用卡。从服务员的角度看，根据账单卖酒也更麻烦，每个人来买酒的时候，他都要先找到买家的账单。当账单太多的时候，事情会变得更麻烦，因为在这么多账单中找到需要的那个还真不容易。这是有状态服务的工作模式。

有状态服务和无状态服务的关键区别是无状态服务的实例是完全可交换的，客户端可以访问任意实例而不用担心结果会有不同。相反，有状态服务需要知道不同请求的一些上下文信息，对于某个请求，并不是每个服务都是可用的。任何时候客户端想要使用一个有状态的服务，都必须黏滞在这个服务器实例上面，以避免产生各种难以预期的后果。

现在让我们看看前端层的主要状态类型，以及如何处理这些状态。

管理 HTTP 会话

所有的 Web 都使用 HTTP 会话。事实上，当你访问一个网站的时候，你的请求就是 HTTP 会话的一部分。由于 HTTP 协议本身是无状态的，Web 应用在 HTTP 基础上创造了一个叫会话的概念，以便服务器可以识别同一个用户的多个请求，并为这些请求创建一个复杂持久的上下文。

从技术角度看，会话是用 cookie 实现的。图 3-3 所示为一个简单的时序图。如果用户不带会话 cookie 发送请求给 Web 服务器，服务器会开启一个新会话，然后通过响应发送一个带有会话的 cookie 给用户。根据 HTTP 协议，cookie 在一组连续的请求响应之间传输。

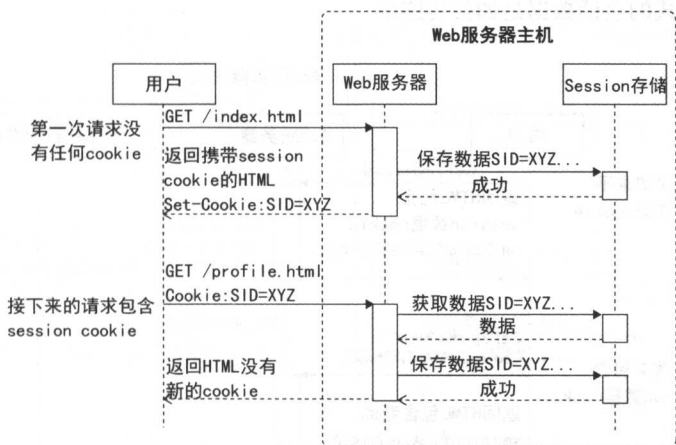


图 3-3 创建 HTTP session

通过使用 cookie，服务器可以识别出来自同一个会话的请求。即使多个浏览器使用同一个 IP 连接到 Web 服务器上，Web 服务器依然可以用 cookie 识别出哪个请求是属于哪个用户的。因此可以实现用户登录功能和其他类似功能。

当你登录到一个 Web 网站的时候，Web 应用通常会在整个会话期间记录你的用户身份及一些其他附加信息。Web 框架或者应用容器负责将会话数据存储在“某个地方”，在整个 Web 会话期间，应用处理每个 HTTP 请求时，都可以访问这些被存储的数据。如果用 Java 开发，会话数据会被存储在 Web 应用容器的内存中；如果用 PHP 开发，默认情况下会话数据存储在本地文件系统中。看到这里，你应该已经想到，这些会话数据需要存储在 Web 服务器之外，这样才能被任何一个 Web 服务器使用到。有三种常用的方法解决这

个问题:

- 将会话状态存储在 cookie 中
- 将会话数据存储在外部数据存储系统中
- 使用支持会话黏滞的负载均衡器

如果你决定将会话数据存储在 cookie 中,事情就非常简单了。应用像平常那样使用会话就可以了;在发送响应给客户端之前,框架序列化会话数据并加密,会话数据会包含在响应头部的 cookie 中。这种方式的主要优点是,你不需要在数据中心存储会话数据,请求每次都会自己带着会话数据到 Web 服务器,这样就保证了应用是无状态的。图 3-4 所示为这种方式的会话数据是如何传递的。

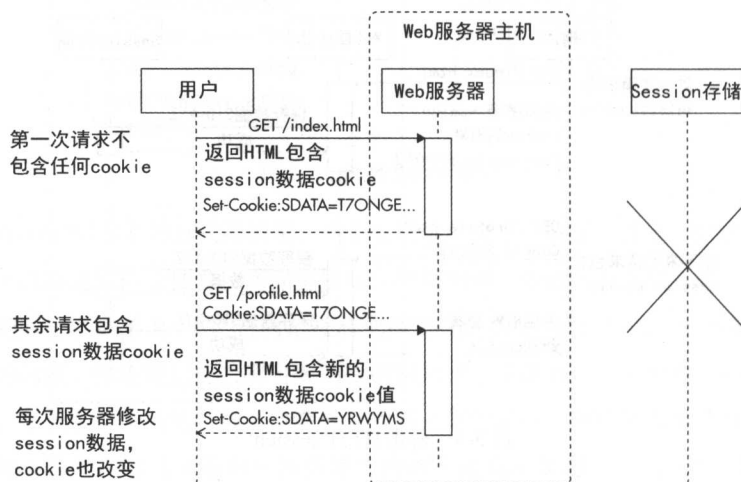


图 3-4 session 数据存储在 cookie 中

使用 cookie 存储会话数据的唯一麻烦是会话存储的代价非常高昂。无论请求的资源类型是什么,请求每次都会发送 cookie,即使是下载图片或者 CSS 文件,或者是发送 AJAX 请求。

只要你保证数据足够小,使用 cookie 作为会话数据存储也可以很好地工作。如果需要存储的会话数据是用户 ID 或者一个安全令牌,这无疑是一种简单又快速的解决方案。不幸的是,通常你一不小心就会加更多的数据到 cookie 中,导致 cookie 变得数以 K 计,请求变得很慢,如果用移动设备的话,会更慢。此外,cookie 数据要加密后再进行 Base64

编码，数据量会增大三分之一，1KB 的会话数据会变成 1.3KB，每次请求和响应要额外传输更多的数据。

第二种方式是将会话数据存储在外部的数据存储服务器。这种方式下，Web 应用不需要通过请求传递会话数据，而是从外部存储服务上加载会话数据。在处理完请求以后，向用户发送响应之前，应用序列化会话数据并将其写入数据存储服务器。这种模式下，Web 服务器不需要在不同请求之间持有任何会话数据，从而使其实现 HTTP 会话无状态。图 3-5 所示为这种场景下如何存储会话数据。

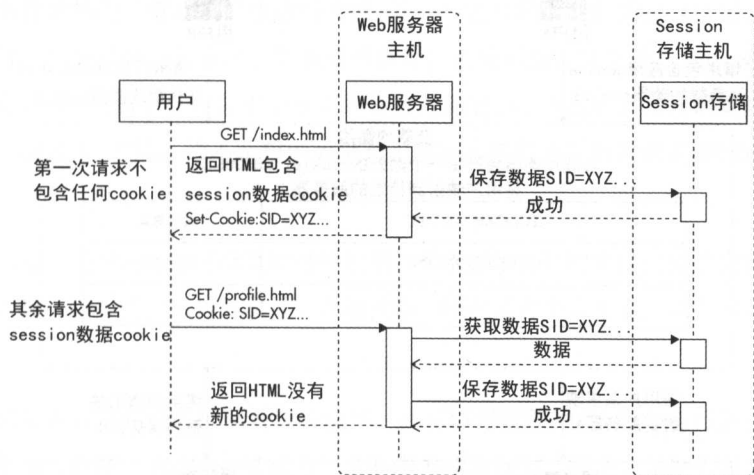


图 3-5 Session 数据存储在分布式数据存储中

很多数据存储引擎都适用于这种模式，比如 Memcached、Redis、DynamoDB 或者 Cassandra，最主要的要求是基于 key 的读写操作的低延迟。最好数据存储引擎能提供自动的伸缩性，如果不能，应用自己做数据切分也不是一件难事，因为可以用会话 ID 切分。我们会在第 5 章进一步讨论数据切分，现在让我们假设会话存储服务器的水平伸缩性是完全没有问题的，可以很简单地用数据分片实现。

如果你用基于 JVM 的编程语言（Java、Groovy、Scala）开发前端 Web 应用，你也可以使用 Teracotta 这类对象集群技术实现会话存储。Terracotta 通过引入同步、分布式锁、一致性保证等机制，提供多台服务器之间的对象访问功能。从前端伸缩性视角看，所有的 Web 服务器都完全一样以方便增加备份实现自动伸缩和水平伸缩。

最后，你可以在应用层不做任何事情去处理 session 状态，而把这个工作交给负载均衡器。这种模式下，负载均衡器需要检查请求 HTTP 头的 cookie 信息，然后将请求发送给最初创建这个 cookie 的服务器上。图 3-6 所示为会话黏滞的一种实现^[L18, L19]。在这个例子中，当用户第一次发送请求时，负载均衡器会将这个请求发送给某个服务器，然后在响应中插入一个负载均衡 cookie，这样负载均衡器就可以跟踪这个用户的请求并将它们都发送给同一个服务器。

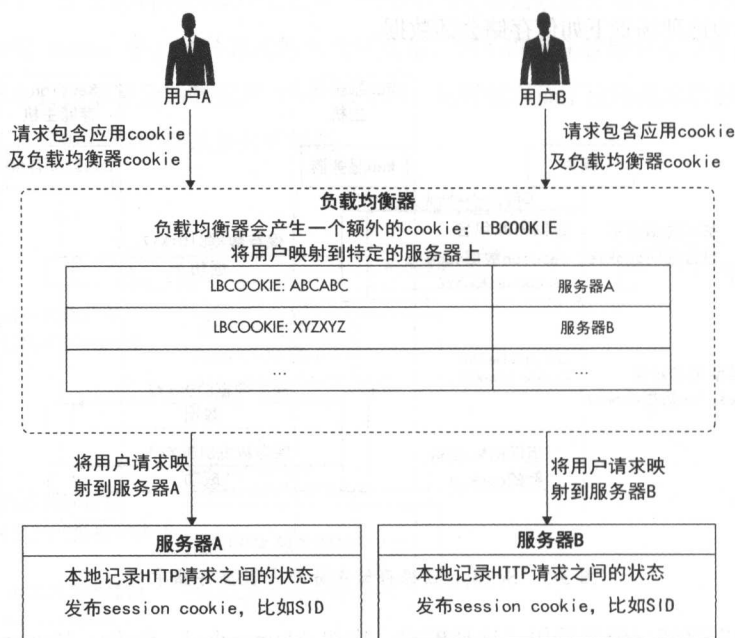


图 3-6 基于附加 cookie 的 session 黏滞

这个方案看起来不错，会话黏滞使得应用无所谓有无状态，但是不推荐使用这个方案。一旦你允许 Web 服务器存储本地数据，服务器之间存在差异，你的系统就失去了弹性。你的服务器没法重启，没法下线，也没办法安全地进行自动伸缩，因为会话数据在特定的服务器上，没办法迁移。另外，使用会话黏滞开了一个危险的先例。如果在 Web 服务器上存储会话数据是 OK 的，那么存储其他数据是不是也 OK 呢？事情一开了头，就没法控制了。负载均衡支持的会话黏滞带来的各种潜在问题远远超过为应用伸缩性带来的好处。实践中，尽量使用前两种方案，即用 cookie 记录会话数据，或者将会话数据存储在一个共享的存储中供所有 Web 服务器使用。

管理文件

Web 应用前端第二常见的状态类型是文件存储。有两类文件需要关注：

- 用户上传到服务器的文件
- 系统生成供用户下载的文件

最常见的情况是用户上传文件分享给其他用户访问。十年前，很少有网站让用户上传照片，而现在图片、视频这类富媒体的需求越来越多，迫使很多 Web 应用不得不考虑在不牺牲伸缩性的情况下管理用户生成的文件。另一种相对少见的情况是让用户下载系统生成的文件，可能是报告、发票、视频或者图片，系统创建这些文件，并生成 URL 供用户下载。有时候，只需要在内存中生成而不需要存储它们，不过更多时候需要将文件存储起来使其不被改变。例如，你不希望因为发布了新版本代码而导致发票的格式或者内容被改变。

文件可能是公开的，也可能是针对特定用户私有的。公开的文件比如社交媒体上的照片，任何人都可以下载。私有的文件，比如发票、报告或者个人消息，就只能被授权的用户访问。

无论你是否在亚马逊云上部署你的应用，都可以考虑使用 S3 服务进行分布式文件存储，当然你也可以使用 Azure 存储服务。这些服务相对便宜，很适合早期开发阶段，或者是使用内部存储不合适的时候。无论你怎样存储文件，都应该使用内容分发网络（CDN）为终端用户提供公开文件分发。公开文件设置一个非常长的失效期，CDN 可以一直缓存这些文件。通过这种方式，存储这些文件的原始服务器只会受到很少的访问流量压力，更容易伸缩。图 3-7 所示为如何存储公开文件及如何通过 CDN 访问。

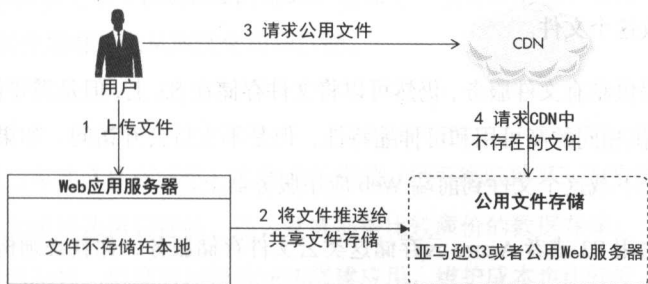


图 3-7 分布式存储及提交公用文件

如果用户上传文件不可以公开访问,那么下载请求就必须通过前端服务器,而不能通过 CDN 访问,图 3-8 所示为一个如何进行这方面配置的例子。一个访问文件的 Web 请求发送给前端 Web 应用服务器后,应用检查用户权限决定是否可以访问该文件,如果访问被接受,那么应用从共享文件存储下载文件并将其发送给用户。

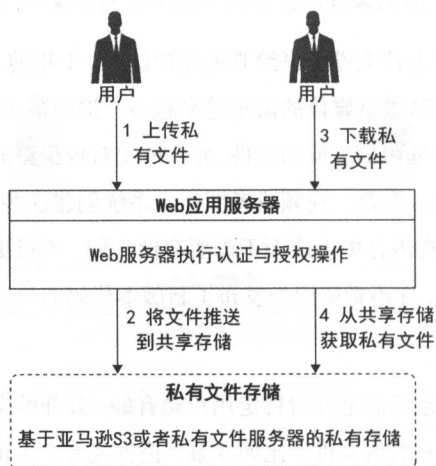


图 3-8 存储与提交私有文件

如果你将应用部署在亚马逊云上,那么 S3 就是最好的文件存储方案。不论存储公开还是私有文件,都可以存储在 S3 上,以保证前端服务器是无状态的。S3 支持公开和私有两种访问方式,可以被公开访问,也可以设置成只被应用服务器访问。

如果你需要提供公开文件服务,只要简单地将文件上传到 S3 公有存储上即可。S3 提供透明的高可用和可伸缩性,你无须操心任何事,S3 会为你搞定一切。你只需要做两件事:上传用户文件到 S3,然后在数据库里记录下公开的 URL 和文件位置,以防将来万一要删除或者修改这个文件。

如果需要提供私有文件服务,仍然可以将文件存储在 S3 上,但是需要使用私有存储。S3 私有存储提供相同的高可用和可伸缩特性,但是不支持公开访问。如果想要访问一个私有文件,要先下载这个文件到前端 Web 应用服务器上。

如果你没法用 S3 或者 Azure 云存储这类云文件存储服务,你就必须构建自己的文件存储和分发方案,你可以选择开源组件,但是你必须自己将它们集成到你的系统中,这可

能需要大量的工作。如果你需要存储大量文件，但是不需要提供很高的访问吞吐量，你可以考虑使用独立磁盘冗余阵列 (RAID)，在多台文件服务器上提供分布式的冗余文件存储。根据具体配置不同，文件服务器提供几 TB 到几十 TB 的存储空间。你还需要考虑高可用性的问题，磁盘级别的冗余并不能满足可用性的要求（要想达到真正的高可用，你需要将一个文件存储到多个物理服务器上，保证服务器级别的可用性）。如果想要对一个文件提供很高的并发读写操作，情况会变得更复杂。你可能需要将一个文件切分成大量的小文件并分布在很多服务器上，或者使用固态硬盘 (SSD)，以提供较高的吞吐能力和较低的随机访问时间。

如果你需要自己实现文件存储的伸缩性，可以考虑上传文件的时候对文件进行分区，随机选择一台服务器存储文件，然后将文件路径存储到一个元数据数据库上。当你需要更多服务器的时候，可以使用加权随机算法选择服务器，将一定百分比的新文件写到每一个节点上。高可用可以通过 RAID 控制器实现，如果需要更高的可用性，可以对文件做备份。你可以在写文件时将文件同时写入两个服务器，也可以利用一个简单的同步机制将文件从主服务器同步到从服务器。

开发一个简单的文件存储是相对容易的，但是如果真的要做到可伸缩高可用就比较复杂了，费时费钱费力。除了你自己做，你还可以选择用开源存储引擎存储文件。比如 MongoDB 允许你使用 GridFS 将文件存储到 MongoDB 集群中。GridFS 是 MongoDB 的一个扩展，可以将文件分割成若干小的块，然后将这些块当作普通文档存储在 MongoDB 里。这样做的好处是可以利用 MongoDB 的分区和复制特性，不需要自己实现。还有其他基于 NoSQL 的数据存储提供类似的功能，比如 Netflix 开源的 Astyanax，它是基于 Cassandra 开发的，利用 Cassandra 的透明分区、数据冗余、失效转移这些核心功能作为底层数据存储。此外，Astyanax 还在 Cassandra 的基础上增加了一些特定的文件存储功能。例如，打乱下载顺序以避免集群热点从而优化访问性能。

提示

记住分布式文件存储是一个复杂的问题。尽可能选择第三方提供商，比如 S3 这样的。如果没法用云存储，那么就选那些比较廉价的数据存储。这些存储也许性能不是特别好，但是可以帮你快速搭建应用，维护成本也比较低。如果这些存

储都不合适，可以考虑自己从头开发一个。如果你决定要自己开发，先研究别人的分布式文件系统是如何做分布式与高可用的，比如 Google 文件系统（GFS）^[w44]，Hadoop 分布式文件系统（HDFS）^[w58]，ClusterFS^[w61, L15]。

管理其他类型的状态

还有一些其他类型的状态会混进你的应用，阻碍你的系统实现可伸缩性，包括本地缓存、应用内存状态、资源锁等。前端应用常常需要缓存一些数据以改善性能、减轻 Web 服务层及数据层的访问压力。第 6 章将讨论缓存的细节。

实时竞标应用是一个对缓存不一致比较敏感的应用。如果你开发一个电子商务网站，需要实时显示竞价，你又想缓存竞价数据以改善性能，那么当价格变化的时候，你就必须使缓存的竞价数据失效。如果这些数据缓存在各个 Web 服务器的内存中，每个数据都存在多个备份，想协调一致使这些缓存同时失效是一件非常困难的事。在这种情况下，你应该使用共享对象缓存，这样的话，每个数据只有一份缓存，使这一个缓存数据失效就容易多了。

图 3-9 所示为多个服务器的同一个数据有不同版本的情况，出现了价格不一致的危险。幸运的是，不是所有的情况都对缓存不一致这么敏感。举个例子，如果你开发一个在线博客系统，比如 tumblr.com 这种，你可以在 Web 服务器上缓存用户姓名和关注者人数，加快页面响应速度。这种情况下，如果访问的 Web 服务器不同，可能看到的关注者人数会不同，但是这种数据不一致通常是可以接受的，不会造成太大影响。

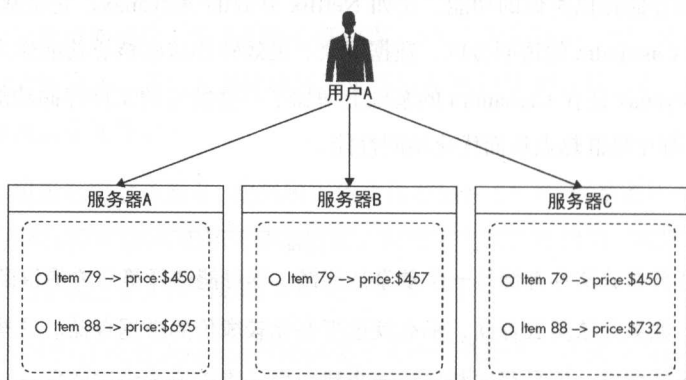


图 3-9 同一个缓存对象有多个副本

最后一个常见的服务器状态的例子是资源锁。锁用来保护并发争用及同步对共享资源的访问。有些时候，人们会在前端层使用锁以保证某些资源的排他性访问。这时，你需要使用分布式锁系统才行。这些年来，我见过很多应用号称自己可以支持水平伸缩，但是却使用本地锁同步共享资源的访问。然而这种锁并没有什么用，它只能在每台服务器上起作用，不可能在服务器之间同步。所以不能在 Web 服务器上使用锁，必须将这种状态“推”到应用服务器之外，就像前面提到的 HTTP 会话数据及文件存储一样。

我们思考一个管理用户 eBay 竞价的应用，看看本地锁是如何限制应用伸缩性的。如果你只部署一台 Web 服务器，那么你可以使用本地锁同步每次竞标的价格。在这种方式下，在同一时刻只有一个线程/进程能够访问同一个竞价。图 3-10 所示为这种部署方式。

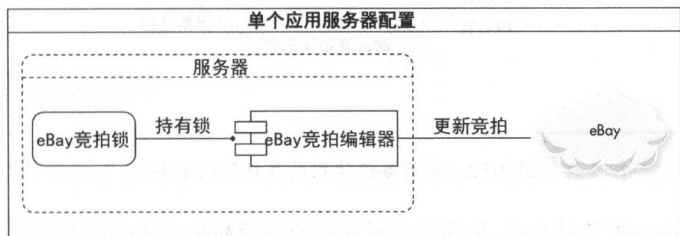


图 3-10 单台服务器使用本地资源锁

如果你增加服务器备份，部署两个服务器实例，如图 3-11 所示，锁就不会如预期那样正常工作。在同一时刻，在服务器 A 和服务器 B 上，有两个并发的线程，同时修改同一个 eBay 竞价，这两个线程并不知道自己修改的是同一个数据。

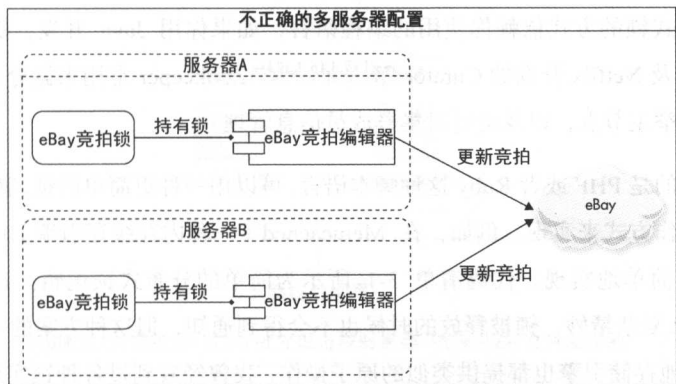


图 3-11 两个备份使用本地/独立锁

为了避免这种情况,需要结合功能分割和增加备份两种伸缩方案。首先将锁功能从应用代码中移出,并创建一个独立的锁服务。然后用这个新的共享锁服务为所有的 Web 应用服务器提供锁服务。通过这种方式,Web 服务器不需要持有本地状态(锁),因此可以增加备份,并随时替换和关闭。图 3-12 所示为这种部署方式。

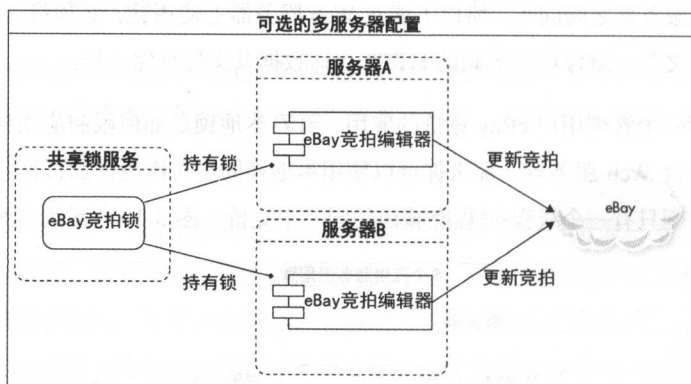


图 3-12 所有备份使用共享锁管理服务

这是伸缩的一种常见方式,将那些全局需要的功能和状态从应用中隔离出来成为一个服务。由于这类功能适用范围很窄很特别,因此比较容易实现伸缩,还能把共享状态从其他系统中分离出来成为一个抽象层。

这种方式的缺点是会增加延迟,因为应用需要执行一个远程调用,而此前本地操作就可以了。而且这样会增加更多组件,也就伴随着管理、伸缩、维护的复杂度变得更高。

实现分布式锁的方式依赖你使用的编程语言,如果你用 Java 开发,那么我建议用 Zookeeper,以及 Netflix 开发的 Curator 库^[47, L16, L17]。Zookeeper 常用来做分布式锁、应用配置管理、选举主节点,以及运行时集群成员信息管理。

如果你用的是 PHP 或者 Ruby 这种脚本语言,可以用一种更简单的锁,就是用 NoSQL 存储的原子操作方式来实现。例如,在 Memcached(一种内存缓存引擎)中,锁可以用一种加操作很简单地实现。代码清单 3-1 所示为简单的分布式锁逻辑。这种方式不像 Zookeeper 那么复杂精妙,锁被释放的时候也不会得到通知,但这种方案的一个好处是很容易伸缩。其他存储引擎也都提供类似的原子操作,我曾经看到过各种锁实现,用 Redis、Memcached,以及 MySQL 和 PostgreSQL 这种关系型数据库实现。

代码清单 3-1 使用 Memcached 实现简单分布式锁

```

$cache->add('lockName', '1', $timeoutInSeconds);
if ($cache->getResultCode() == Memcached::RES_NOTSTORED) {
    // some other process has the lock
}else{
    // I got the lock
}

```

总之，保证所有的 Web 服务器都是无状态的，包括所有的 Web 前端服务器和 Web 服务服务器。保证服务器无状态可以让你很容易添加机器实现伸缩。下一节中，我们将详细剖析每一种前端组件，查看它们的伸缩性影响，讨论如何利用前端服务器的无状态特性进行自动化伸缩。

可伸缩的前端组件

现在我们看看前端基础设施中每一个组件的伸缩性影响，以及每个领域使用的技术实现。图 3-13 所示为前端层常见关键组件的概览图。

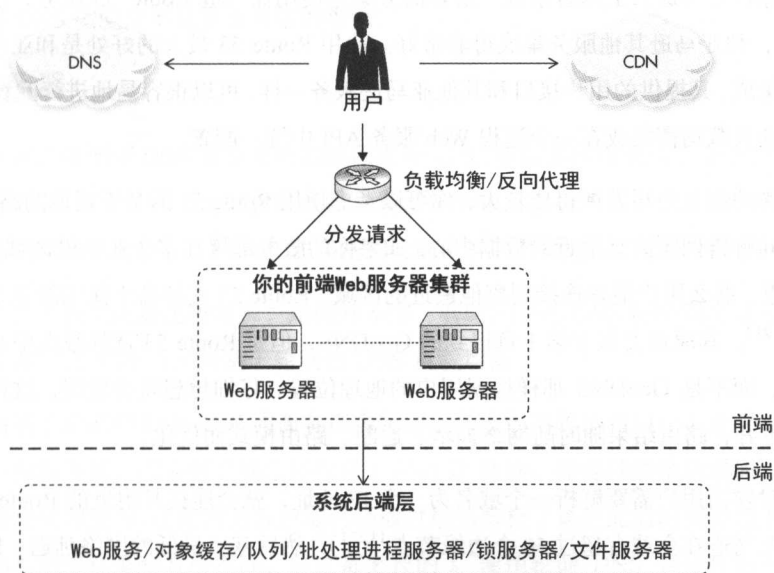


图 3-13 前端基础架构详情

前端层包含的组件有 Web 服务器、负载均衡器、域名系统 (DNS)、反向代理, 以及 CDN。前端层的组件主要负责与用户建立连接和用户交互。下面我们详细讨论每一个组件。

DNS

当用户访问网站时, 他们第一个需要交互的组件是域名系统 (DNS)。无论是直接访问一个 Web 网站还是用移动 APP 连接一个 Web 服务, 用户在连接到服务器之前, 都需要找到服务器的 IP 地址。本质上, DNS 就是用于将 ejismont.org 这样的域名解析成 173.236.152.169 这样的 IP。

考虑各种情况, 我推荐使用第三方的域名服务, 而不是部署自己的 DNS 基础设施。除非你有非常特别的需求, 否则不要自己搞。假如我自己就是一家主机服务商, 需要支持的 DNS 条目成千上万, 那我会考虑部署自己的 DNS 服务器, 以获得更好的弹性, 节省更多的钱。市面上大的 DNS 服务商有几十家, 找一家便宜、可靠、伸缩性好的不是什么问题。

如果你在亚马逊云上部署系统, 最好的方案是使用亚马逊 Route 53 服务, 这是一个 DNS 服务, 和亚马逊其他服务集成得非常好。使用 Route 53 最大的好处是和亚马逊生态系统无缝集成, 其提供的用户接口和其他亚马逊服务一样, 可以很容易地进行配置。例如, 可以与弹性负载均衡集成在一个远程 Web 服务 API 中统一配置。

如果你的创业公司发展得比较大, 你可以考虑使用 Route 53 的基于延迟的路由功能, 将用户访问解析到距离其最近的数据中心。如果你的服务部署在多个亚马逊区域 (多个数据中心) 里, 那么用户最好连接到离他最近的区域。Route 53 支持易于使用的基于延迟的路由^[L20~L21]。其原理类似于第 1 章提到的 GeoDNS, 但是 Route 53 选择数据中心是基于响应延迟, 而不是 GeoDNS 那样根据用户的地理位置。仔细想想就会发现, 这种技术比 GeoDNS 更好, 路由结果随时随网络拥堵、通断、路由模式而优化。

任何时候当用户需要解析一个域名为一个 IP 地址, 就会连接其附近的 Route 53 DNS 服务器 (亚马逊在全球有超过 50 个边缘节点^[L22])。然后基于最低的网络延迟, Route 53 服务器选择一个负载均衡器的 IP 地址响应给用户 (要看哪个区域离用户最近)。图 3-14 所示为这种路由的执行过程。应用部署在两个亚马逊区域, 一个在欧洲, 另一个在北美。

这种情况下，从古巴访问的用户可能会得到欧洲区域也可能得到北美区域的 IP 地址，具体要看访问哪个区域延迟最小。

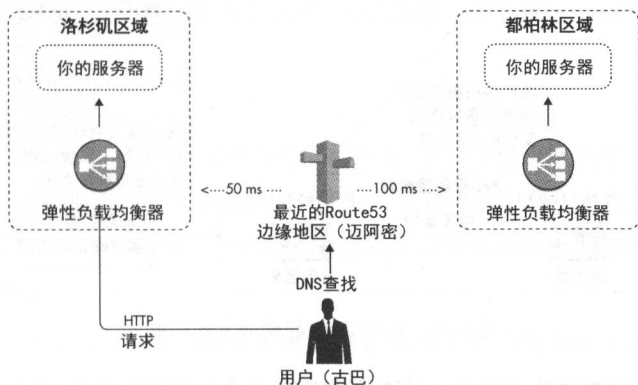


图 3-14 基于延迟的 Route 53 路由

如果你没有将应用部署在亚马逊云上，还有其他很多供应商可以选择，比如 easydns.com、dnsmadeeasy.com、dnsimple.com，还有 dyn.com。这些 DNS 服务商提供相近的服务等级、延迟和正常运行时间保障^[L23~L24]，换一个供应商通常也不是一件难事，所以选择 DNS 供应商不是重点考虑的事情。

负载均衡器

一旦客户端调用 DNS 服务将域名解析成 IP 地址，就需要连接到这个 IP 地址请求 Web 页面或者 Web 服务。强烈建议使用负载均衡器作为数据中心的入口点，这样做一则进行伸缩扩容的时候更方便，二则改变底层基础设施架构的时候也不会影响到用户。

过去，负载均衡器比较昂贵，用的并不多，有时候会用 DNS 将访问流量分发到多个 Web 服务器。图 3-15 所示为基于轮询的 DNS 负载均衡。

使用基于轮询的 DNS 负载均衡有很多问题，最大的问题是对用户不透明。你没法随便移除一台服务器，因为客户端已经缓存了它的 IP 地址。你也没法做到增加一台服务器就立刻提高处理能力，还是因为客户端已经通过域名解析了缓存此前的服务器 IP 地址（缓存时间随不同缓存策略有所不同，总之都挺长的）。使用轮询 DNS 直接将访问流量分发到 Web 服务器也使得服务器管理和失效恢复变得格外麻烦，我不建议在生产环境使用这种方法。

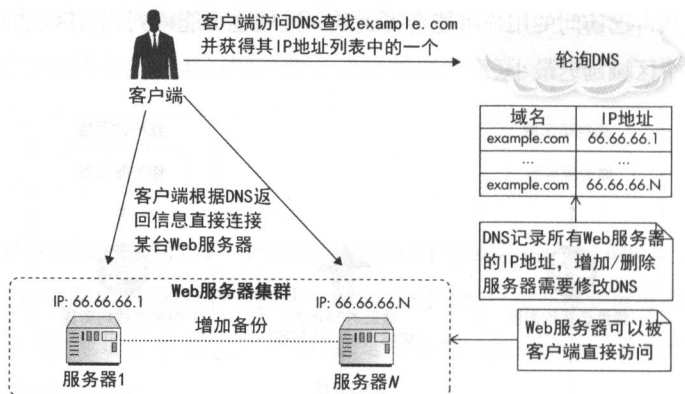


图 3-15 基于 DNS 轮询的负载均衡

实践中可行的方法是，在 Web 服务器和客户端之间配置一个负载均衡器，如图 3-16 所示，Web 服务器和客户端之间的所有流量都会经过负载均衡器。这样，你的整个数据中心的结构和服务器的职责都对客户端隐藏起来了。

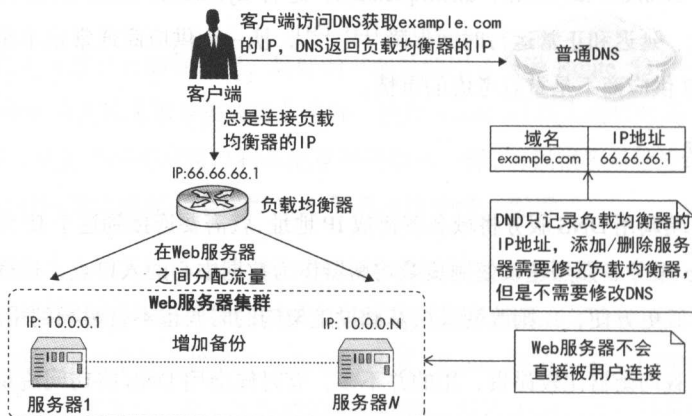


图 3-16 基于负载均衡器的部署

使用负载均衡器还有一些其他好处。

• 隐藏服务器维护

你可以将一个 Web 服务器从负载均衡池移除，等这个服务器上所有的请求都处理完了，就可以安全地关掉这个服务器而不会影响任何一个客户端。你可以使用这种方法实现“滚动更新”，在整个集群更新软件的版本而不会造成任何不可用事故。

- 无缝增加处理能力

任何时候都可以增加更多 Web 服务器，而且不会影响用户的体验。一台新的 Web 服务器一旦添加到负载均衡池中，就可以立刻接受用户连接，不像 DNS 负载均衡那样会有传播延迟。

- 高效的失效管理

将失效的 Web 服务器简单地移出负载均衡池，就可以很快地处理 Web 服务器失效的故障了。比加一台服务器花的时间更短，立即就可以将一台已经宕机的服务器从负载均衡池中移除出去，这样新的连接就不会分发给这台失效的机器。

- 自动化伸缩

如果你的数据中心基于云服务部署，并且配置了自动化伸缩（比如亚马逊云、Open Stack，或者 Rackspace），你就可以在一天中根据流量随意增加或者减少 Web 服务器。通过负载均衡器，可以将这种增减过程自动化，不会引起任何不可用或者对用户产生任何影响。本章后面部分将解释自动化伸缩。

- 高效资源管理

使用安全套接层（SSL）卸载可以降低 Web 服务器的资源消耗。SSL 卸载，有时候也被称为 SSL 终结，是一种负载均衡功能，即在负载均衡器上处理所有的 SSL 加密解密工作，从而在数据中心内部全部使用非加密的连接。我个人推荐使用 SSL 卸载，尽快摆脱 SSL 制约。

如你所见，使用负载均衡器作为数据中心的入口有很多好处。负载均衡器在网站开发领域非常流行，有大量的工具和技术可以选择。由于每个系统都是不同的，所以很难在不知道应用细节的情况下推荐一个具体的技术。目前的系统主要使用的负载均衡类型大概有三种，让我们快速过一下这三种技术，并探讨相关的可用性技术问题。

使用云主机的负载均衡

像很多创业公司一样，如果你选择将应用部署在亚马逊 EC2 或者 Azure 上，那么我强烈建议你使用他们提供的负载均衡服务，而不是部署自己的负载均衡器。比如亚马逊提供的弹性负载均衡（ELB）。ELB 是一种“负载均衡即服务”，由亚马逊管理，提供可伸缩的负载均衡服务。只需在 Web 控制台上配置一下，将一组 EC2 实例关联起来，就可以轻松使用 ELB 了。使用 ELB 的好处有：

- ELB 是开始阶段最便宜最简单的方案，因为用户几乎没有什么组件需要管理和伸缩。
- ELB 支持透明伸缩，所以不必担心负载均衡器成为瓶颈。
- ELB 具有内置的高可用性，所以不必担心 ELB 成为失效的单点。如果决定安装部署自己的负载均衡器，要确保其具有自动的失效转移能力，在主负载均衡器失效的时候，需要有热备的负载均衡器随时准备切换提供服务。
- ELB 的成本收益比很好，可以按需付费，启动一个 ELB 实例也几乎没有什么前置成本。
- ELB 集成了自动伸缩功能，如果有 EC2 实例失效，可以自动替换。本节后面会讲述关于自动伸缩的内容。
- ELB 可以实现 SSL 终结，所以从 ELB 到 Web 服务器的连接将是 HTTP，而不是 HTTPS（基于 SSL 的超文本传输协议）。因为 Web 服务器不需要再处理 SSL，这可以很大程度降低 EC2 实例的资源需求。
- ELB 支持后端服务器的优雅关闭。你可以将一台 Web 服务器从负载均衡器移出不终止已经建立的连接。你可以从负载均衡池移出某台服务器，等已经连接的客户端都完成处理断开后，再安全地关闭这个服务器实例，而不会对用户产生任何影响。
- ELB 可以完全使用亚马逊 SDK 进行管理，以你期望的方式对负载均衡器进行配置变更。比如，你可以自动在多台机器上部署，代码部署期间，服务器实例会自动从负载均衡池中移出。

如你所见，ELB 是一个非常强大的备选方案。亚马逊这些年为 ELB 开发了许多功能，比过去更好用。如果说 ELB 有什么不适合你的应用的原因，可能只有一个：

- ELB 需要一定的伸缩“预热”时间。如果你的网站流量突然爆发，需要几秒到几分钟内将流量处理能力翻一番，ELB 对你来说也许太慢了。ELB 在自动化伸缩方面非常强大，但是如果你的访问流量突然爆发，它无法很快提高自己的伸缩处理能力。这种情况下，某些用户可能会收到 503 错误响应，直到 ELB 伸缩性扩大到能够处理这些新增加的流量。

除了面向外网访问的负载均衡器，某些云服务商，比如亚马逊和 Azure，还提供面向内网的负载均衡。图 3-17 所示为面向内网的负载均衡。在这种使用场景下，你可以在前端服务器和内部服务之间部署负载均衡。如果来自 Web 服务器的 Web 服务请求全部都通过内部负载均衡器，那么你就可以在数据中心内部享受负载均衡的各种好处。你可以增加服务器提升处理能力，可以在服务器维护期间从负载均衡池移除它们，可以在多台服务器之间分摊请求压力，可以提供自动化的失效转移。

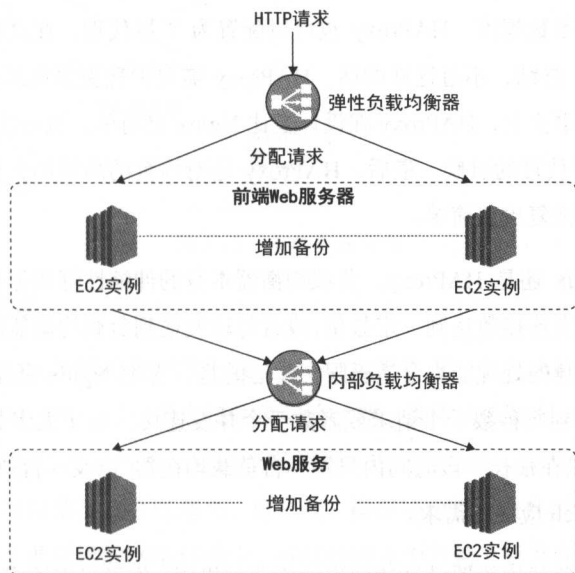


图 3-17 内部负载均衡器

软件实现的自管理负载均衡器

如果你使用的云服务商不提供负载均衡服务，或者提供的服务不符合你的需求，那么你需要一个开源（软件实现的）负载均衡器。你可以用反向代理比如 Nginx，也可以用专门的开源负载均衡产品比如 HAProxy。市面上有很多产品可以选择，不过这两个是最常用的。

Nginx 的最大优点是它同时也是一个 HTTP 反向代理，可以缓存来自 Web 服务器的 HTTP 响应。这个特性使得 Nginx 成为内部负载均衡的首选。你不但可以通过往 Nginx 池中添加服务器实现 Web 服务的伸缩，还能利用 Nginx 的缓存功能，减少 Web 服务层的资

源需求。第 4 章和第 6 章将讨论更多这方面的内容。现在,你只需知道 Nginx 是一种非常好的反向代理/负载均衡方案。

HAProxy 就比 Nginx 简单多了,仅提供负载均衡功能。它可以配置为 4 层或者 7 层负载均衡器。当 HAProxy 设置为 4 层代理的时候,它不会查看更高层的协议内容,仅仅依赖 TCP/IP 头进行流量转发。反过来说,HAProxy 可以针对任何协议进行负载均衡,而不只是 HTTP/HTTPS 协议。你可以利用 HAProxy 分发各种服务的访问流量,比如缓存服务、消息队列或者数据库。HAProxy 也可以配置为 7 层代理,在这种模式下,可以实现会话黏滞和 SSL 终结,不过这样的话,HAProxy 需要消耗更多资源去探测跟踪 HTTP 有关的各种信息。事实上,HAProxy 在设计上比 Nginx 更简单,执行上比 Nginx 更轻量,特别是配置成 4 层代理的时候。最后,HAProxy 还有内置的高可用支持,处理失效的时候更有弹性,失效恢复也更简单。

无论你用 Nginx 还是 HAProxy,负载均衡器本身的伸缩性都要你自己搞定。当连接到负载均衡器的并发连接数达到一定数量,或者每秒发送到负载均衡器的请求达到一定数量,负载均衡器自身的处理能力会达极限。幸运的是,无论 Nginx 还是 HAProxy,可以处理的极限都能达到每秒数千个请求或者数千个并发连接。对于大多数应用而言都足够了,所以你应该可以在很长一段时间内只用一台负载均衡器(以及一台热备的负载均衡器)就可能满足你的 Web 应用的需求。

如果只用一台负载均衡器已经达到你的应用的极限,你可以部署多台负载均衡器,这些负载均衡器各自有自己的公开 IP,通过配置轮询 DNS 将流量分发给它们。图 3-18 所示为如何用这种方式对一台负载均衡器实现伸缩扩容。

如你所见,利用 DNS 实现负载均衡伸缩并不麻烦。你的负载均衡器是可互换的,你的 Web 服务器是无状态的,那么你就可以添加更多负载均衡器实现水平伸缩。管理多个负载均衡器有点小复杂,配置变更需要在多台负载均衡器上同步,但是这依然是一种相对简单的伸缩性方案。

配置轮询 DNS 到多个负载均衡器要比到多个 Web 服务器更合适,因为你不可能在负载均衡器上运行业务逻辑。你不需要像对待 Web 服务器那样升级或者重新部署负载均衡器,负载均衡器也很少会因为 BUG 而失效。

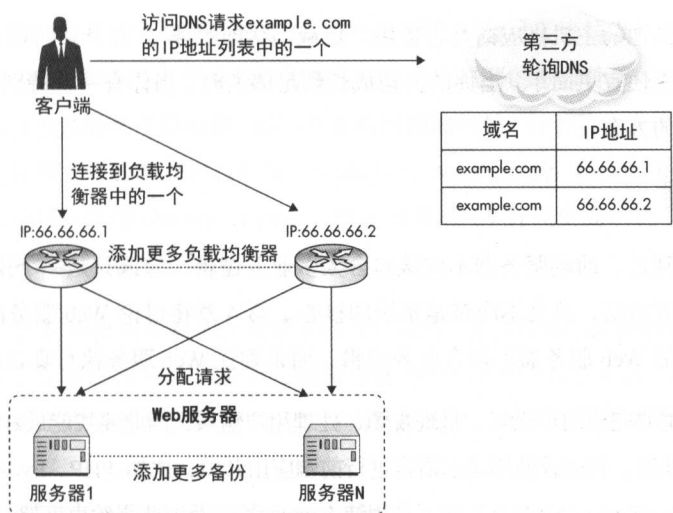


图 3-18 多级负载均衡

硬件负载均衡器

最后，处在伸缩性顶端的是硬件负载均衡器。如果你有一个自建的数据中心，需要处理非常高的访问流量，那么可以考虑硬件负载均衡器。这样的设备有 F5 的 Big-IP，Citrix 的 Netscaler，相比软件负载均衡，它们能提供非常丰富的功能及非常强的处理能力。使用硬件负载均衡，可以获得高吞吐能力，低延迟，操作一致性。硬件负载均衡器经过专门的优化，安装在一个低延迟的网络环境中。可以处理几十万到上百万的用户连接，进行垂直伸缩也很容易。^[L25-L26]

硬件负载均衡有一个显而易见的缺点是：太贵了。硬件负载均衡器的价格低得要几千美元（很低端的那种设备），高的要十万美元。硬件负载均衡器的另一个麻烦是使用它们需要经过特别的培训，很难找到有相关工作经验的工程师。总而言之，如果你有一个高并发流量的网站，部署在自己的服务器硬件上，同时你又不觉得贵，硬件负载均衡器是最好的选择。

我相信“负载均衡即服务”在将来会变得更普遍。这是一种更一般化的解决方案，大多数网站也都需要这样的服务，云主机平台不提供这样的服务就显得不完整。事实上，除了亚马逊云提供的“负载均衡即服务”，还有 Azure 负载均衡器支持内网外网负载均衡，Rackspace 的云负载均衡器，Open Stack 的 Lbaas。所以即使你不在亚马逊云上部署应用，

你也可以看看其他的主机供应商是否提供“负载均衡即服务”。在开始阶段，通过他们的服务进行伸缩更便宜更简单。当你的公司成长到足够大时，当你有一些特别的需求时，再考虑使用自己的方案。

Web 服务器

我前面提到过，前端服务器不应该包含太多业务逻辑，应该只是一个用来集成 Web 服务层结果的展示层，总之不应该是系统的核心。第 4 章将讨论 Web 服务的优点，现在我们就假设前端 Web 服务器不包含业务逻辑，而是委托 Web 服务执行真正的工作。

由于前端主要完成用户交互、呈现视图、处理用户输入，因此要找能最好地完成这些任务的技术进行开发。我推荐使用动态语言进行前端应用开发，比如 PHP、Python、Groovy、Ruby，甚至是 JavaScript(Node.js)，而不是用纯 Java 或者 C 及那些强约束框架，例如 Java EE、JSF 或者 CGI 开发。你使用的技术栈要能很容易地解决那些常见的前端问题。比如，你需要简单又快速地处理 SEO、AJAX、国际化，以及每日模板变更。系统的各个层使用相同的技术栈有很多好处，可以很容易地进行代码复用，整个技术团队也只需要掌握较少的技术。话虽如此，同一个系统的前端服务器和后端服务器分别使用不同的技术也并非不常见，不同的层面面临的挑战并不相同，使用不同的架构风格也有好处。

选择了一种编程语言和开发框架后，还需要选择你的应用运行在哪种 Web 服务器上。幸运的是，从伸缩性视角看，选择何种编程语言、使用何种 Web 服务器并不重要。只要前端服务器是无状态的，就可以简单地通过增加机器实现水平伸缩。

Node.js 是一个运行时环境，也是一组开发组件，允许开发者使用 JavaScript 在 Web 服务器端进行开发。这是一门相当新的技术（2009 年才开始开发），却被许多赞誉包围，有许多激动人心的概念。在长时间没太多通信的情况下，以及在用户端和服务端之间只需要交互很小的数据包的情况下，它可以维持几万到几十万的并发用户连接。在这类应用中，运行 Node.js 的服务器支持的用户量要远远超过其他技术。

可能有人会说：选择不同的服务器差别大了去了，Node.js 能处理几十万的并发连接，而 Apache 撑到几千并发就不行了。我的答案是：是又不是。说是，确实对于某些场景，

有的技术是比另一些技术有更好的伸缩性，不过，这并不会对最终结果产生多大影响，正如我曾经说过，整个集群的水平伸缩能力要远远比一台机器的垂直伸缩能力更重要。所以从一开始就关注整体的水平伸缩性，而不是某些特别的场景。对于某些应用，比如聊天室、即时通知、支持多人对战游戏的后台等，Node.js 比 Apache 或者 Tomcat 更合适，但是对于其他 98% 的应用，使用 Groovy、Python、PHP 或者 Ruby 开发更简单更容易，这些语言有更庞大更成熟的社区和生态环境。

市面上有很多 Web 服务器和开发技术栈可以选择，完全看你团队的经验和个人喜好。在决定选择某个技术栈和 Web 服务器之前需要做一些调研的工作，不过，如我之前所说，不论你选择什么样的 Web 服务器，对伸缩性而言最重要的是保证前端机器的无状态。

提示

在做技术选型调研时，不要想当然地猜测，也不要完全凭信那些所谓的评测结果。评测就像是做民意调查，结果难免带有人为的痕迹。试着猜测一下那些评测结果中欲言又止的东西。要想从评测结果中获得更多信息，要理解评测的究竟是什么，如何做的评测，约束条件是什么。最后，多关注图表，里面可能有很多误导人的东西^[L27~L28]。

缓存

对于 Web 应用的前端伸缩性而言，缓存是最重要的技术之一。使用缓存不是为了加入更多的前端服务器或是让这些服务器更快响应用户请求，缓存是为了避免让前端服务器处理请求。事实上，缓存对 Web 应用伸缩性至关重要，第 6 章将专门讲述。为避免重复，这里仅简要讨论和应用前端伸缩性相关的几个组件。

你要做的第一件事情是集成 CDN。第 6 章将详细讨论 CDN。CDN 可以作为所有 Web 请求的代理，也可以仅仅作为静态文件的代理，比如图片、CSS，以及 JavaScript 文件。

如果你决定让所有的访问流量都经过 CDN，那么你可以缓存整个页面甚至是 AJAX 响应。对于某些 Web 应用类型，你可以只用 CDN 就处理了绝大部分访问流量，减少了服务器的负载压力并提高响应速度。

坏消息是，不是所有的 Web 应用都能使用 CDN 有效缓存整个页面。很多 Web 内容

都是个性化的，还有一些内容是动态的，很难将整个 HTTP 响应都缓存了。在这种情况下，更好的办法是部署自己的反向代理服务器以控制哪些内容要被缓存及缓存多久。常用的反向代理有 Varnish 及 Nginx，具体内容将在第 6 章展开。

前端层缓存的另一种手段是直接将数据缓存在浏览器。现在的浏览器支持 Web 存储可以保持相当多的数据（数以 MB 计）。这种特性在开发移动客户端及 SPA 应用时特别有用，在更新用户界面时请求的数据量尽量少。通过在浏览器端存储 JavaScript 代码，可以提供更流畅的用户体验，同时减轻服务器端的负载压力。

最后，如果请求不能通过浏览器缓存或者反向代理缓存完成，Web 服务器就需要处理这些请求并生成响应。这时，Web 服务器仍然可以使用对象缓存将部分响应片段缓存起来。绝大多数情况下，应用都可以使用一些共享对象缓存组件，比如 Redis 或者 Memcached。事实上，很多创业公司通过增加 Memcached 集群就能做到每天服务上百万的用户。使用 Memcached 改善伸缩性的例子有很多：Facebook^[w62]、Pinterest^[L31]、Reddit^[L32]，以及 Tumblr^[L33]。

自动伸缩

自动伸缩是一种自动化管理网站基础设施的技术，根据访问压力及服务器负载增加或者减少虚拟服务器的数量。伸缩性有两个方面，一方面是伸，即需要扩容，另一方面是缩，即需要缩容，缩容主要是为了节约成本。自动伸缩不属于前端组件技术，不过在整体 Web 技术栈中，自动伸缩更容易在前端实现，并且带来的收益也格外大。

为了更好地理解自动伸缩的重要性，我们看一下图 3-19。图中展示的指标数据是什么不重要，重要的是负载压力呈现的周期模式（这张图来源于一个免费的 ISP 监控工具）。我们看到每天的访问流量有显著变化，周末也会有明显不同。你可以根据负载压力手动添加或者移除虚拟机，不过最好是能够自动完成这个过程，系统监控自身的压力和负载状况，自动扩容或者缩容。根据系统访问压力，使用自动伸缩可以节省大概 25%~50% 的 Web 服务器主机成本。此外还能帮你在没有人工干预的情况下处理突发的访问峰值。

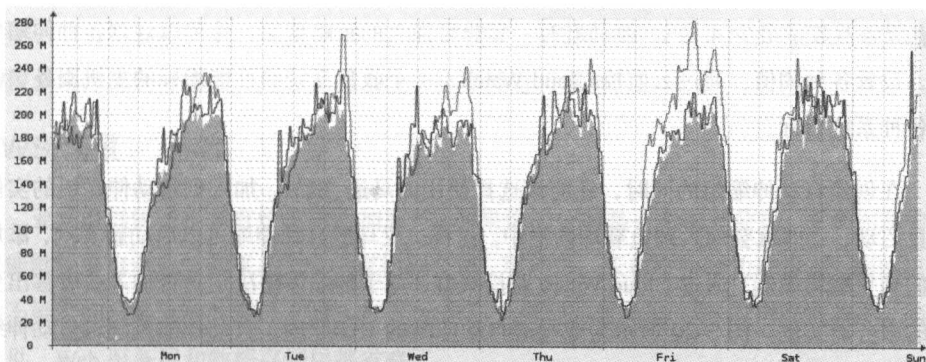


图 3-19 计算资源使用模式

实现自动伸缩最简单的方式是使用主机供应商提供的自动伸缩工具。在试图自己实现自动伸缩之前先看看主机供应商有没有提供这类功能。亚马逊作为云主机服务商的领导品牌已经实现了“自动伸缩即服务”，其他供应商比如 Rackspace 及 Azure 现在也提供自动伸缩功能作为云主机套餐的一部分。为了更好地理解自动伸缩的工作原理及包含了哪些组件，我们看一个亚马逊的例子。

首先，要想配置自动伸缩，Web 服务器必须使用 EC2（弹性计算云）实例。当使用自动伸缩功能的时候，就可以在一天中的任何时候添加或者移除服务器。自动伸缩可能会在任何时间点移除服务器，所以不能在服务器上存储任何数据，或者至少要保证存储的数据是一次性的（比如缓存）。关闭服务器不会导致注销用户或者中断用户操作的情况。

在自动创建 EC2 之前，你需要先创建一个 Web 服务器镜像（亚马逊机器镜像[AMI]），然后配置为可以自启动并且可以自动加入到集群中。要做到这一点，需要将一个新的 EC2 实例变成一个可以提供完整功能服务的 Web 服务器，所需的一切都在 AMI 文件中，具体配置可以通过 AMI 启动参数获得，或者通过远程数据存储获得。亚马逊允许服务器镜像带启动参数，这样你可以创建一个新的实例并告知它属于哪个集群或者服务器角色是什么。你也可以使用亚马逊存储服务，比如 SimpleDB，存储 EC2 实例的启动参数，当通过某个 AMI 镜像启动一个 EC2 实例时，可以从共享存储中加载必要的配置参数并将自己配置成为一个拥有完整功能的 Web 服务器。

接着，可以创建一个自动伸缩组定义伸缩策略。自动伸缩组是 Web 服务器的逻辑表述，它有这样一些策略，“当 CPU 使用超过 80% 的时候加两台服务器”，或者“每天上午

9 点服务器数量最少 4 台”。亚马逊有一套非常强大的策略框架，你可以规划各种伸缩事件，设置各种阈值，通过云监控 Cloud Watch（一个收集系统级性能指标的主机服务）收集各种系统指标。

在创建自动伸缩组的同时，还应该选择使用亚马逊 ELB。加入到自动伸缩组的实例一旦启动，立即就会加入到负载均衡池中。这样，亚马逊自动伸缩可以启动新实例，添加它们到负载均衡器，通过 Cloud Watch 监控集群，基于伸缩策略决定是否加入或者移出更多的服务器实例。图 3-20 所示为亚马逊自动伸缩的工作原理。自动伸缩控制着自动伸缩组中所有的机器实例，随时更新 ELB 将服务器加入或者移出集群。

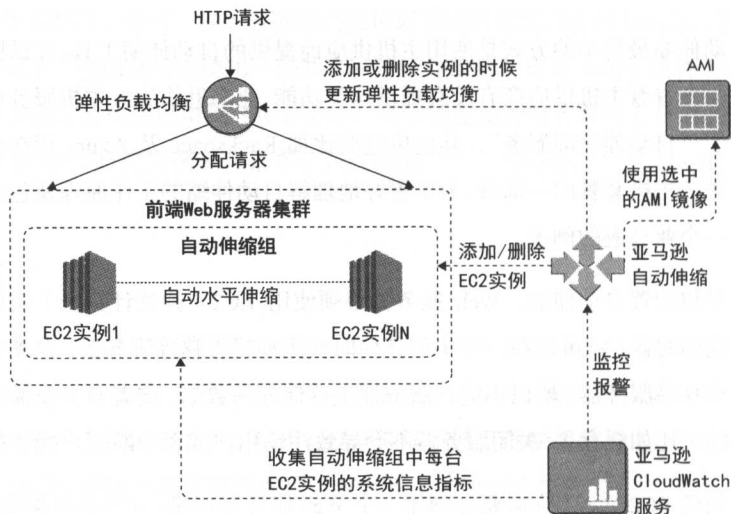


图 3-20 亚马逊自动伸缩

自动伸缩在某些方面有点像第 2 章提到的自动恢复，系统可以自己处理问题而无须人工干预。无论你有什么台服务器，无论访问高峰何时到达，都不需要你的运维工程师去监视系统负载决定是否要调整集群规模。系统可以根据当前状况自己决定是否调整基础设施，进而以一种低成本的方式（低谷的成本）提供一个良好的用户体验（高峰的服务）。

部署案例

最后，我们把这些组件放在一起，看下两种不同的部署场景：使用亚马逊云部署 Web

应用；使用私有数据中心部署 Web 应用。当然，这里描述的仅仅是一个蓝图，里面包括的很多组件都是可选的，可以根据系统实际需要进行裁剪。

AWS 场景

亚马逊提供了很多有价值的附加服务。如果你的公司是一个初创公司，那么你真的需要能够快速启动运转起来。作为一家初创公司，可能每一天都在生死边缘徘徊，面临巨大的不确定和总是紧缺的资源。图 3-21 所示为一个典型的亚马逊 Web 应用部署蓝图，简化起见，Web 服务层和数据存储层都省略了。

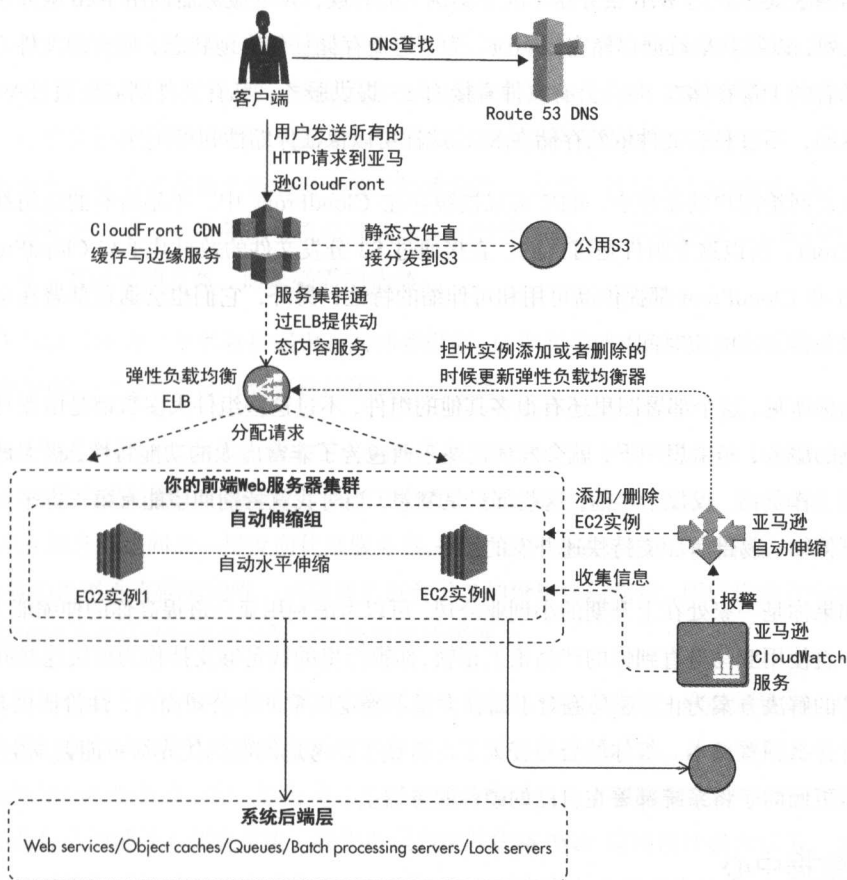


图 3-21 亚马逊部署示例

通过这张图可以看到亚马逊是一个支持全栈部署的公司。对部署一个可伸缩的 Web

应用需要用到的所有东西都考虑到了。图 3-21 中唯一需要自己负责的就是 EC2 实例，即便如此，亚马逊还是相应地提供了检查 EC2 实例失效，关闭宕机的服务器，基于自动伸缩需求创建新实例的功能。

在图 3-21 的例子中，应用使用 Route53 作为 DNS。由于 Route53 提供即买即用的高可用和可伸缩特性，你无须担心 DNS 的管理和伸缩。这张图的下面，Web 请求到达 ELB，这里你可以实现 SSL 卸载，并通过轮询方式将访问流量分发到自动伸缩组的每台服务器上。当然，你也无须担心负载均衡器的可用性和伸缩性，ELB 对这些特性同样即买即用。

当请求最终到达 Web 服务器（EC2 实例）的时候，Web 服务器调用 Web 服务层、缓存、队列，以及共享数据存储完成响应。为了避免存储任何本地状态，所有的文件（公有的和私有的）都存储在 S3。公有文件直接由 S3 提供服务，私有文件则需要通过 Web 服务器返回，不过私有文件依然存储在 S3，这样可以保证伸缩性和可用性。

在返回给用户的途径中，响应可以被缓存在 CloudFront 中。不是所有的应用都需要 CloudFront，所以这个组件是可选的。直接通过 S3 分发文件的效果几乎与 CloudFront 一样。S3 和 CloudFront 都提供高可用和可伸缩的特性，另外，它们也会通过部署在全球的边缘服务器加速响应时间。

如你所见，这个部署图里还有很多其他的组件，不过这些组件大多数都是由亚马逊直接提供的服务，稍微想一下，就会发现这些东西包含了非常庞大的功能特性，极大地解放了你的工作负担。仅仅十年前，这些都只是梦想，当时我就特别期望能有第三方平台提供这种可伸缩、易部署、支持快速开发的服务。

如果你是一家处在上升期的小创业公司，可以考虑利用亚马逊提高你的伸缩能力。你可以一直使用亚马逊直到你的产品走上正轨，你银行里的钱足够支持你为解决这些问题定制自己的解决方案为止。亚马逊对于面临大量不确定因素的小公司而言，性价比极高，几乎没有什么预置成本。等你的公司变大了，有钱了，考虑问题的优先级可能会发生变化，你可能更倾向于将系统部署在自己的硬件服务器上。

私有数据中心

第二种部署场景是基于物理数据中心的自购硬件。这种配置下，比较容易使用的第三方服务只有 DNS 及 CDN。也有人在使用自建数据中心的同时使用 S3 作为文件存储，不

过这种用法并不常见。

虽然在裸机上部署系统需要你自己管理大部分事情，但是使用自购硬件也有一些好处。在自己的硬件上部署系统的主要好处有：

- 你可以对延迟与吞吐有更多的预测把握能力。在自己的硬件上部署可以使服务器与服务器之间的通信延迟达到亚毫秒的级别。
- 物理服务器比虚拟服务器的处理能力更强大。从云上迁移到裸机上需要的服务器数量会减少。
- 对于小公司而言，预付一大笔钱买服务器可能不太合算，但是一旦你的网络工程师团队成长起来了，管理的服务器规模达到上百台，自己买服务器就比租“计算单元”便宜。有些东西，比如内存、输入输出（I/O）操作、SSD 硬盘，在云上还是蛮贵的。一般情况下，使用自购硬件的时候，垂直伸缩更高效。
- 有些公司需要遵循严格的安全或者法律标准。比如，有些关于赌博网站的法律条款就规定所有的服务器主机必须放在某个特定的地方，这样的话，自购硬件就不是可有可无的方案，而是必需的选择。

图 3-22 所示为一个私有数据中心的部署模型。这里我仍然推荐使用第三方的 DNS 及 CDN 供应商，但是其余部分就要你的团队自己管理了。

跟亚马逊部署方式类似，请求首先到达负载均衡器，一个硬件设备：HAProxy 或者 Nginx。如果你还需要一个缓存层，你可以使用 Nginx 作为负载均衡器，或者在负载均衡器与 Web 服务器之间放一层反向代理服务器。通过这种方式，你可以缓存整个 HTTP 响应。负载均衡器垂直伸缩的唯一问题就是每台设备的价格比较昂贵，所以你或许应该使用 DNS 轮询的方式将访问流量分发到多台负载均衡器上。

由于你自己不可能按需提供硬件，所以也就不能实现自动伸缩，你必须更小心地协调规划伸缩计划。添加硬件需要几周的时间，如果公司官僚作风严重，甚至需要几个月，所以仔细规划你的服务器容量，因为没办法在访问高峰时点一下鼠标就能添加新机器。不过即使你在自己的硬件上部署系统，我仍然强烈推荐你将 Web 应用设计成无状态。采用这种方式，尽管你还是没办法自动伸缩，但是至少不会成为水平伸缩的障碍。

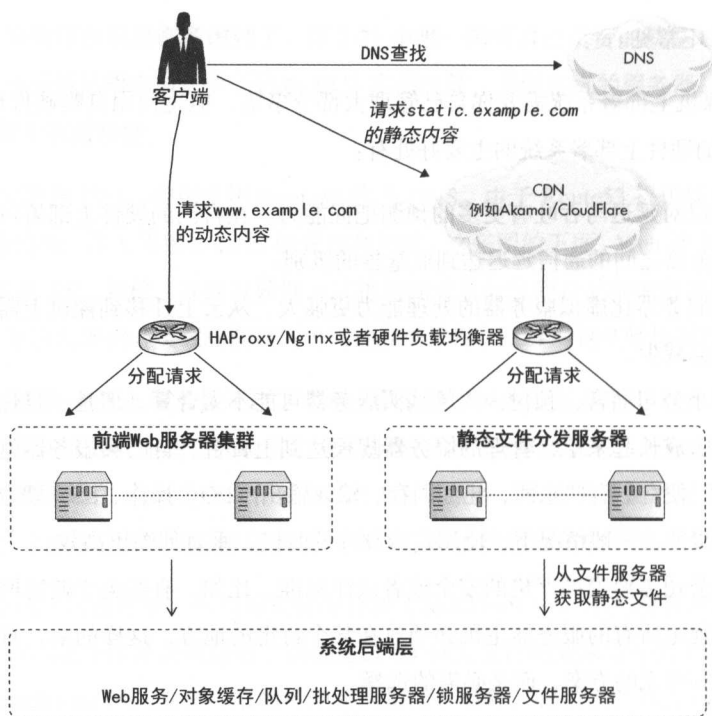


图 3-22 私有数据中心部署

在自己的硬件上部署应用，依然需要考虑如何部署共享文件存储。文件存储方案的选择主要看吞吐量和文件大小，本章前面曾经提到过几种方案。我倾向的是那种应用可以无须知道文件如何被存储及被替换的方案。根据具体预算和需求，你可选的方案包括 FTP 服务器、SAN，以及 NoSQL 数据存储。

无论选择哪一种文件存储方案，你都应该通过 CDN 对外提供文件服务。如果使用 S3，公有文件会自动提供 HTTP 访问方式，所以你可以使 CDN 指向这些文件。对于私有数据中心，你需要在文件存储之前加一层 Web 服务器，这样可以通过 CDN 的方式公开访问文件。

如你所见，两种场景，基础设施组件和架构底层原则都是一样的——唯一的不同就是使用的技术不同，要达到的目的是相同的。我觉得在开始时使用云服务更便宜更快速，等你的公司成长得足够大时，可以考虑迁移到私有数据中心上。

小结

一般情况下，前端是需要高伸缩性的关键区域，需要支撑相当高的并发和请求压力。幸运的是，只要设计适当，这个区域的伸缩性相对比较简单，只有少量的状态需要管理（复制或者同步）。

我强烈建议阅读更多关于本章提到的各种技术的书和文章。关于缓存、负载均衡，以及一般的前端性能优化技术有许多相当不错的书籍^[8,48-49]。我也建议阅读一点流行的 Web 框架技术，比如 Spring^[14] 或者 Grails^[22,34]，它们会改善 Web 应用的架构。最后，我建议了解熟悉云主机服务^[29,W34-W36,W38]。

4

Web 服务

Web 服务层需要仔细设计，这是实现主要业务逻辑的地方。在开始讨论 Web 服务实现之前，需要想清楚你是否真的需要 Web 服务，必须要做出哪些取舍。使用 Web 服务有很多好处，比如在较高的抽象层面提供复用性，不过也会带来一些坏处，比如更高的开发成本及更大的复杂性。

为了帮助读者更好地做出决策，我将结合成本与收益，综合探讨各种设计与开发 Web 服务的手段。我也会提到各种伸缩性考量，以及构建可伸缩 Web 服务的最佳实践。在深入可伸缩 Web 服务细节之前，我们先看看各种设计方式。

Web 服务设计

最开始，Web 应用使用一种简单的单一的架构方式设计开发。所有的交互都用超文本标记语言（HTML）及基于超文本传输协议（HTTP）的 JavaScript 完成。大概从 2006 年开始，通过提供各种类型的应用编程接口（API）和 Web 应用交互的方式逐渐流行起来了。这就使得公司可以通过 Web 整合所有的系统。随着 Web 变得越来越大，集成和重用这些 API 的需求就越强烈。到 2010 年前后，移动开发大潮袭来，使得 API 变得更加重要和流行。仿佛一夜之间，几乎所有人都在开发移动 APP。显然，移动 APP 仅仅是换了一

个用户接口而已，和那些已经存在的 Web 应用不论功能还是数据都完全一样。移动应用的流行帮助 API 变成 Web 开发的一等公民。现在，我们就看一看 API 设计的各种方式。

Web 服务作为一种备用表示层

在 Web 应用开发的上下文中，开发 Web 服务的最古老方式是先开发 Web 应用，然后添加 Web 服务作为一种备用的接口。在这种模型中，Web 应用是一个整体，在应用的顶端提供各种扩展接口访问各种数据和功能，这些扩展接口可能不需要 HTML 及 JavaScript。

为了更好地解释这种方式，我们看一个例子。假设要开发一个酒店预订网站，首先要开发前端（混合 AJAX 和 CSS 的 HTML 视图），以及业务逻辑（后端代码通常运行在一个 MVC 框架上）。网站允许用户做一些常规的操作，比如搜索酒店、寻找空房、预订房间等。

开发完这些核心功能，如果还有其他需求出现，这个 Web 应用就需要增加 Web 服务了。例如，产品运行几个月后，你想和一个合作伙伴的系统集成，方便他们预订你的酒店。这种集成就需要你设计实现 Web 服务，允许合作伙伴执行一定的操作，比如基于价格搜索酒店和空房。图 4-1 所示为这个系统的大概架构。

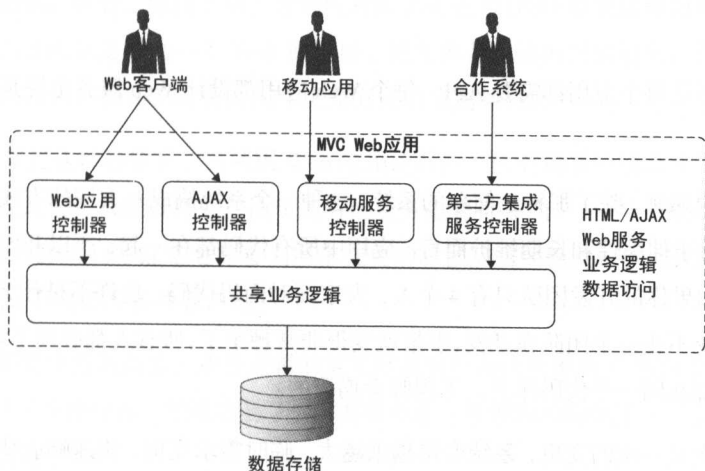


图 4-1 使用 Web 服务扩展的单一大应用

如图 4-1 所示，整个 Web 应用的开发和部署都在一个整体内运行。当然，这并不意味着你就不能把这个应用复制到多台服务器上同时运行，只是说，整个应用是一个进程，

运行在同一套代码上，从代码到进程实例，都没有把展示层和服务层分离开来。事实上，在这种方式里并没有一个明确的 Web 服务层，Web 服务是一整个 Web 应用的一部分。

这类 Web 应用通常使用 MVC 框架开发（诸如 Symfony、Rails、SpringMVC），Web 服务是 MVC 组件的一部分，允许用户无须通过复杂的 HTML/AJAX 方式访问系统。

也许你会说，这种方式很粗糙甚至完全过时了，但还是有一些理由支持人们选择这类方案。这个方案的主要好处是在代码层面快速增加新功能及快速应对变化，特别是在开发的早期阶段。没有 API 也就没有那么多的组件和层次，同时也减低了系统的总体复杂性，开发起来就会更容易。如果你没有用户，你其实不知道自己的业务模型是否正确，如果你想创业早期尽快将一个最小规模的可运行产品放出去让大家试用，那么这种轻量级的开发方式就挺适合你。

这种方式的另一个好处是，你可以延迟实现各种 Web 服务的代码，直到你证明了这个产品确实是可行的，然后再开发 Web 服务。虽然今天的技术使得开发 Web 服务的效率越来越高，但还是会有一些前置成本。例如，你将整个应用当作一个整体来开发，就可以在代码里使用本地对象，而无须增加新的 Web 服务功能。管理 Web 服务的约定及对 Web 服务 Debug 也是一件比较费时的事，这在项目早期是需要认真考虑的，甚至会对项目成败产生影响。

最后，不是每个应用都需要 API，每个 Web 应用都设计 Web 服务也许是一种不必要的过度设计。

不过话说回来，除了那种最简单的系统，这种一个系统搞成一大坨的方式还是比较糟糕的，特别对于伸缩性和长期维护而言。应用中所有代码都在一起，所以开发和部署都必须在一起。如果你的开发团队只有 4 个人，大家面对面码代码，也许不是什么大问题，但是当这个系统不止一个团队在开发，事情就变得非常棘手了，所有人都必须了解整个系统，所有的变更都在同一个代码库上，工程师会崩溃的。

经过一次又一次的变更，系统变得越来越大，临时需求变更、快速响应变更逐渐变得不再重要。反而是依赖隔离、构建高层抽象变得更重要。你要根据自己的情况判断权衡到底哪个更重要。

如果你决定使用单一大系统的方式，你需要小心未来潜在的代价，没准儿将来需要重

构甚至重写。在第 2 章讲过,在进行可伸缩设计的时候,控制耦合及功能分割是非常重要的。幸运的是,单一大系统不是设计应用的唯一方式。让我们看看另一种极端:API 优先方式。

API 优先方式

API 优先设计这个词还比较新,不同人的定义略微不同。我不太同意:API 优先即优先设计编写 API 约定,然后基于 API 实现 Web 服务,再基于 API 开发客户端程序调用 API 及其实现。我比较接受的说法是:只要预先定义好 API 约定,无论先开发客户端还是先实现 API 都无关紧要。

API 优先的概念来源于如何解决多用户接口的问题。通常情况下,一家公司既有移动应用,又有 PC Web 站点,还有移动 Web 站点,甚至还需要和第三方集成,为他们提供程序访问接口,访问系统的数据和功能。

如果分别实现这些用户接口,那么看起来就像图 4-2 所示。在相同的业务逻辑上,散落着一堆不相干的交互组件。由于 Web 应用、移动客户端、合作伙伴每个都有一些不一样的需求,所以每个的实现都有那么一些不同。你也自然会将相同的代码在不同的交互组件之间复制粘贴。然后,你就不得不分别应对需求变更和 BUG 修复这样的麻烦。针对这些麻烦的解决办法就是创建一个 Web 服务层,把主要业务逻辑封装起来,将复杂性隐藏在 API 约定的背后。图 4-3 所示为 API 优先方式的应用架构。

在 API 优先这种场景下,当调用 Web 应用的时候,所有的客户端都使用相同的 API 接口。这种方式会带来很多好处。由于所有的业务逻辑都在同一个 Web 服务中,只需要维护一份代码就可以了。这也意味着当需求变更的时候只需要修改较少的代码,只需要修改 Web 服务而不需要改各种客户端。

另一个需要注意的点是,业务逻辑的复杂性从客户端代码推到了 Web 服务层。反过来说,也就是开发修改客户端更容易了,不需要考虑业务规则和数据库——要做的只是知道怎么调用 API 接口。

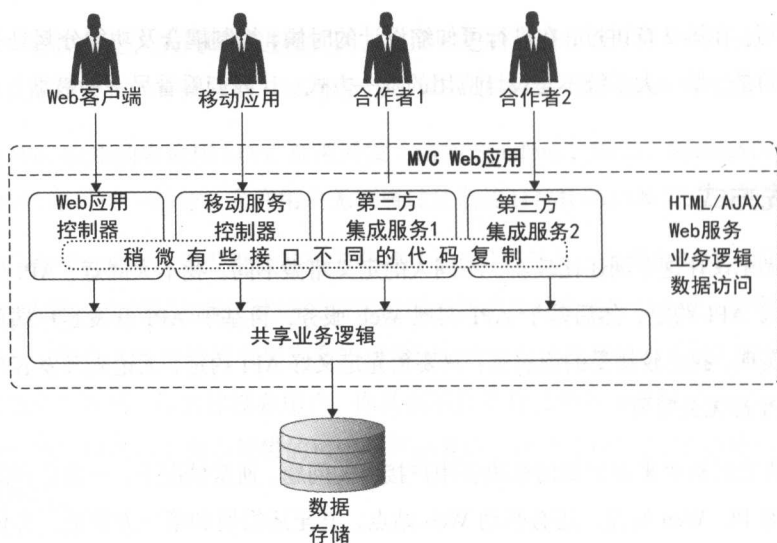


图 4-2 存在多客户端并且有代码复制的应用

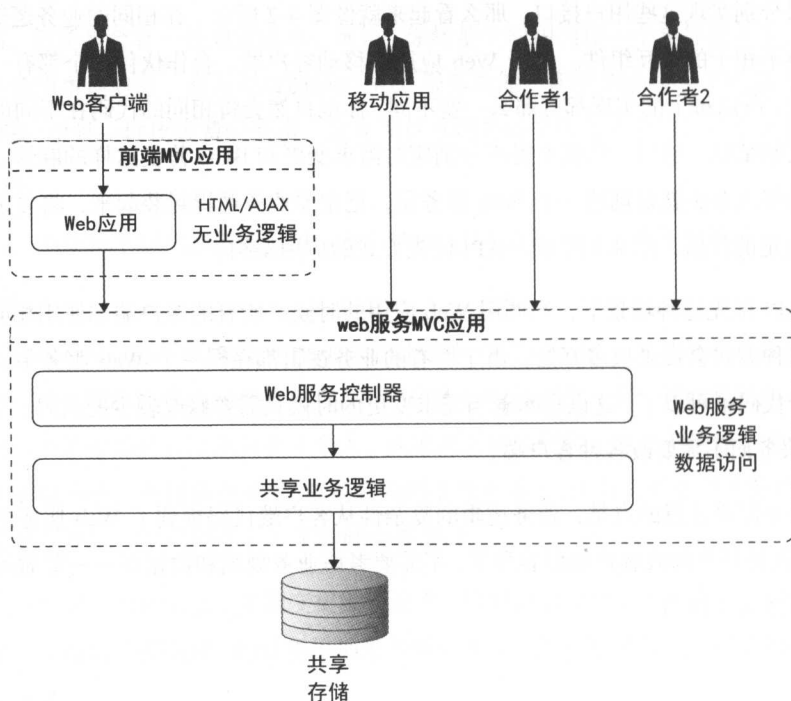


图 4-3 多客户端的 API 优先应用

使用 API 也使得系统伸缩更容易，你可以进行功能分割，将 Web 服务分割成一组更小的独立的服务。通过 API 约定，把客户端和 Web 服务的耦合分解开来。这种解耦合有助于更好地理解系统，如果你是一个客户端开发人员，无须知道 Web 服务是如何实现的，反过来，如果你是 Web 服务开发人员，也无须知道客户端是如何实现的，无须知道他们调用你暴露的 API 接口究竟干了些什么。

从伸缩性角度看，关注点分离有助于客户端和服务端独立伸缩。允许你用更多的服务器分担负载，以及不同的服务使用不同的技术实现，不同的服务部署在不同的服务器上以适应不同的需求。

不幸的是，API 优先的方式说起来容易做起来难。要确保不会过度设计，同时又要提供全部客户端需要的功能，你需要仔细研究需求用例及花更多时间去设计。但是无论你多么努力，仍然会出现有的地方设计过多，有的地方又考虑不周的情况。主要原因是要先设计 API，这时还没有办法足够了解用户需求。

API 优先不是万能的，有的应用适合，有的应用未必适合。我觉得 API 优先的方式更适合成熟的系统和稳定的公司，早期创业公司未必合适。用这种方式开发也许对于软件开发本身而言比较清楚利落，但是需要更多的计划、更多的需求分析、更多的开发资源、更多的设计可伸缩有弹性的 Web 服务的经验。

务实的方式

除了用上述提到的两种方式构建应用，我推荐一种更务实的方式，即上面两种方式的结合。我建议从一开始就考虑用 Web 服务层及面向服务的架构，但是只有等真的必要的时候才去实现它。

这句话的意思是，如果你发现某个需求很容易就分离出一个独立的服务，而且这个功能又会被多种客户端调用，那么你就可以考虑将其服务化。最好拿一些相对比较独立的需求试水，用一些小东西快速学习，而不是一上来就搞个大新闻。

我们举个例子讨论下具体如何操作。假设我们需要在一家刚开始创业的公司开发一个 Web 应用——比方说开发一个自拍编辑网站——那么一开始我们最好先尽可能快地搞一个原型出来。最好先花几个星期的时间验证概念，而不是立马深入到设计、建模、开发 Web 服务和客户端这些细节中。绝大多数的所谓创业点子其实都是伪需求，等你真的开

发出来产品放到用户面前时,才发现他们根本没有兴趣,最好的情况也是他们需要的东西和你开发的东西有一定出入,你必须做出某些变更。除非你能证明人们愿意为你开发的应用付出他们辛辛苦赚来的钱,否则你极有可能是在浪费时间进行过度设计。

话说回来,如果我们的创业公司拿到了几百万美元的投资,或者开发的产品已经有强付费用户的基础,我们需要为这些用户开发一款支持产品,那么我可能会倾向于用 API 优先的方式。例如,如果我们开发的是一个电子商务网站,我们需要开发一个商品推荐引擎推荐其他购物网站的商品,那么最好用 API 优先的方式,开发一个 Web 服务将这个推荐引擎的复杂性隐藏起来。保证系统的稳定性,为商业决策增加信心,确保产品易维护可伸缩,要比渐进式学习更重要。通过将推荐逻辑封装到 Web 服务内部,我们可以提供一个简单的 API 很容易地将这个功能集成到 Web 应用中。而且,无论最开始网站是不是用 API 优先的方式开发,都可以成为这个服务的客户端。一旦我们构建了一个解耦合的推荐引擎服务,就不必关心客户端是怎么组织的。

不幸的是,如果你选择了这种混合模式,就会陷入各种权衡与自我怀疑之中——自己到底在过度设计还是在制造混乱。这种混合模式的结果,就是最终开发出来的系统紧密结合了两部分东西:实现一小部分价值的 Web 应用,实现更大价值和更明确需求的一组 Web 服务。理想情况下,随着时间的推移,公司越来越成熟,可以逐步将这些混乱的原型分离出去,逐步过渡到面向服务的架构。这种方式效果不错,但是过程中还是会经历一些混乱。这种做法听起来有点奇怪,不过基于现有的状况做出最佳决定要比遵循一个严格的规定更合适。

设计 Web 服务时,你还需要选择你打算实现的 Web 服务的类型,进而选择你的架构风格。我们看一看都有哪些 Web 服务类型。

Web 服务类型

Web 服务的设计和实现细节向来都是一个热门话题。我希望你以一种开放的心态面对各种方案,避免陷入教条之中。这里讨论两种主要的 Web 服务架构风格。在讨论每一种类型的时候,我都会从伸缩性和开发速度两个方面探讨其优缺点,但最好还是你自己做出判断,究竟哪种风格更适合你的 Web 应用。我们先看一下以功能为中心的架构风格。

以功能为中心的服务

以功能为中心的 Web 服务由来已久——事实上，可以追溯到 20 世纪 80 年代。以功能为中心的 Web 服务是指能够调用远程机器上的功能或者对象方法，无须知道这些功能或者对象是如何实现的，以何种语言编写，以及运行在什么样的架构上。

思考以功能为中心的 Web 服务的一个简单方法是，想象你的代码在任何地方都可以调用一个功能（任何功能）。作为这个功能调用的结果，你的参数和需要的数据全部都要序列化，然后通过网络传输到一台机器上，在那里操作被最终执行。参数和数据到达远程服务器后，会被那台机器转换成本地数据格式，接着调用功能，执行结果会被序列化，然后通过网络发送到你的调用服务器，再被反序列化成本地机器可以处理的数据格式，最后你的代码接着继续执行，甚至无须知道这个功能调用是在一台远程服务器上完成的。

这种方法理论上听起来挺不错，但在实践中，实现起来其实比较困难，调用要跨编程语言，跨 CPU 架构，跨运行时环境，要每一方面都遵守严格而精确的参数约定、变量转换，以及错误处理。此外，还要面对各种其他的挑战，比如资源锁、安全、网络延迟、并发、接口升级等。

曾经有好多以功能为中心的技术，比如公共对象请求代理体系架构（CORBA），可扩展标记语言——远程过程调用（XML-RPC），分布式组件对象模型（DCOM），简单对象访问协议（SOAP），这些技术关注的都是如何让客户端代码调用一个在远程机器上实现的功能。不过这么多年过去了，经过开发的洗礼和标准演化，目前可以说 SOAP 一枝独秀。这种局面部分源于 SOAP 的可扩展性，部分源于几家最大的科技公司的支持，比如 IBM、Oracle、Sun、BEA，以及微软。

SOAP 最常见的实现方案是使用 XML 作为描述和消息编码方式，通过 HTTP 在客户端和服务端之间传递请求和响应。SOAP 最重要的一个特性是允许 Web 服务基于自身的约定描述完成服务发现并生成集成代码。

图 4-4 所示为 SOAP 的工作原理。首先，Web 服务提供者暴露一组 XML 资源，诸如 Web 服务定义语言（WSDL）文件、XML 模式定义（XSD）文件、WSDL 描述方法与可用的终端信息，XSD 描述请求与响应中用到的数据结构。这些资源文件就是 Web 服务的约定，客户端代码如何生成，Web 服务如何调用，这些必要的信息全部都在这些资源文

件中。举个例子，如果你用 Java 开发，那么你可以用专门的工具和库下载资源文件，然后生成 Java 本地客户端代码。这些代码就是一些 Java 类，可以被编译并用在应用程序中。在这些代码的背后，会依赖到 SOAP 相关的一些库，封装了数据序列化、身份认证、路由、错误处理等各种实现细节。客户端代码无须知道自己调用的是远程 Web 服务，只需要简单依赖基于 Web 服务约定（WSDL 和 XSD 文件）生成的 Java 库就可以了。

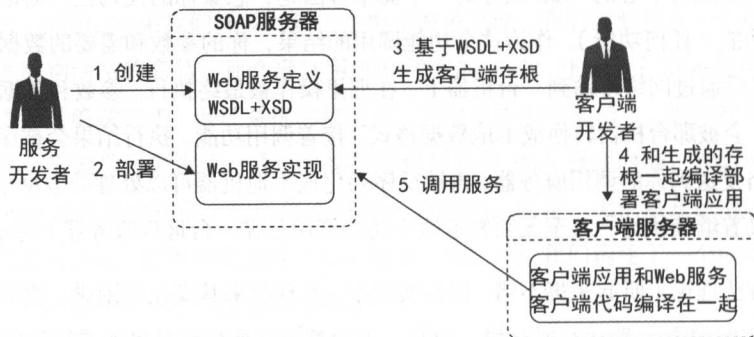


图 4-4 SOAP 集成流程

SOAP 设计方面的另一个重要特性是其扩展性。过去几年间，创建了一大堆附加的规范，集成了一些高级特性，诸如事务、支持多阶段提交，以及多种身份认证和加密格式。事实上，这些规范非常之多，人们将这些规范统一以 ws-*命名（ws-context、ws-coordination、ws-federation、ws-trust，以及 ws-security）。带来的麻烦就是，不同技术开发的系统之间集成变得更困难，不同的 Web 服务提供者对规范的支持程度不同，支持的版本也不同。

在 Web 开发领域，开发者喜欢使用动态语言，比如 PHP、Ruby、Perl，甚至 Python，这些语言和 SOAP Web 服务集成会比较困难。使用这些技术开发客户端代码没有问题，但是集成的时候会有麻烦。这些技术开发 SOAP Web 服务也不太实用，相关的支撑工具和开发库都太少了。Web 技术似乎被 SOAP 世界排除在外，没有一家巨头尝试实现或者做一些支持的工作。所以，Web 需要一种 SOAP 的替代技术，可以更容易地集成，更方便地实现。这就使得 JSON——基于 REST 风格的服务流行起来。

提示

能够发现服务及进行明确的约定是 SOAP 的牛逼之处，所以我希望能够很容易地构建 SOAP 服务。不幸的是，由于动态编程语言这方面工具和库的缺乏，用

这些技术构建 SOAP 服务并不实际。我用 Java 构建 SOAP 还可以，但是用 PHP 的时候，我确信真的不行。

在某些场景下，SOAP 的互操作性和易用性是考量的重点，有些场景下，伸缩性更重要，而 SOAP 没办法在 HTTP 层面做缓存。SOAP 请求是通过 XML 发送的，请求参数和方法名都在 XML 里面。URL 里面不包含远程过程调用需要的全部信息，响应也就没办法在 HTTP 层面基于 URL 进行缓存。使用 SOAP 会使应用的伸缩性变差，应用没办法像通常那样使用反向代理进行缓存。

SOAP 对伸缩性而言比较严重的另一个问题是，某些附加的 ws-* 规范会在 Web 服务协议上引入状态，使 Web 服务变得有状态。理论上，你可以只用 SOAP 相关规范的一个最小集实现一个无状态的 SOAP Web 服务。实际中，这些规范用着用着就失控了。一旦你开始支持事务及安全会话之类的特性，你的 Web 服务也就告别了无状态，没办法在不同机器之间分发请求了。

虽然 SOAP 带来比较高的复杂性，以及有一些伸缩性方面的缺陷，但我们还是可以在某些方面学习和借鉴，比如严格的约定、数据类型，以及功能发现。如果你的业务主要是企业应用方面，比如银行、保险之类，那么 SOAP 的一些优点，比如安全、分布式计算功能等，就比较合适。但是，我不觉得 SOAP 是一种适合开发可伸缩 Web 应用的技术，特别对于创业公司而言。SOAP 已经不再是 Web 服务开发的主流技术，除非老板逼你用，否则没必要用 SOAP，开发的复杂性和维护的工作量相当大。

幸运的是，SOAP 有替代方案，下面让我们看看。

以资源为中心的服务

开发 Web 服务的一个替代方案就是将服务的关注点从功能转移到资源上。以功能为中心的 Web 服务，每个功能都需要若干个参数，产生若干个结果值；以资源为中心的 Web 服务，每个资源都代表一类对象，这些对象上面只能执行有限的操作（可以创建、删除、更新、获取资源）。资源的模型可以是任意的，但是和资源的交互是标准的。

REST 是一个面向资源架构风格的方案，早在 2000 年初就已经提出来。此后，由于其简单性和轻量级的开发模型，逐渐变成 Web 应用集成事实上的标准。

为了更好地理解如何使用 REST, 我们考虑一个在线音乐网站的例子, 用户可以搜索音乐, 针对喜欢的音乐创建公开的播放列表, 将播放列表分享给朋友。如果你想把它做成服务, 可以考虑暴露一个 REST API 给客户端实现搜索歌曲及管理播放列表。你可以创建一个“播放列表”资源, 允许用户创建、获取、更新他们的列表, 针对列表中的歌曲, 还可以再创建一些附加的资源。

需要注意的一个重要事实是 REST 服务使用 URL 统一定义资源。如果你知道一个资源的 URL, 你就可以决定使用某个 HTTP 方法使用这个资源。表 4-1 所示为使用“播放列表”资源的 HTTP 方法一览。通常, GET 方法用于获取资源及其子资源的信息, PUT 方法用于替换资源, POST 用于更新资源或者增加资源条目, DELETE 用于移除对象。

表 4-1 播放列表资源可用的 HTTP 方法

示例 URL: <http://example.org/playlists/324>

HTTP 方法	结果行为
GET	获取用户 324 创建的播放列表
PUT	替换这个用户创建的整个播放列表集合 (你提交一个列表集合)
POST	根据列表名称创建一个新的播放列表
DELETE	删除用户 324 的所有列表

当你使用 POST 方法请求 `/playlist/324` 资源创建一个新的播放列表的时候, 你就为用户 324 创建了一个新的播放列表。这个新创建的列表可以通过 GET 请求发送给相同的 URL 地址 `/playlist/324` 去访问, 这是用户播放列表的父资源。表 4-2 所示为如何和资源 `/playlist/324/my-favs` 进行交互, 为用户 324 创建一个叫作 `my-favs` 的音乐播放列表。

表 4-2 选中的播放列表资源可用的 HTTP 方法

示例 URL: <http://example.org/playlists/324/my-favs>

HTTP 方法	结果行为
GET	获取用户 324 添加到“my-favs”列表中的所有歌曲
PUT	替换整个“my-favs”播放列表 (提交一个歌曲 URL 集合)
POST	添加新的歌曲到播放列表 (提交歌曲的 URL 添加到“my-favs”)
DELETE	删除整个“my-favs”播放列表

这个 API 也可以暴露一些其他附加的资源——一首歌、一张专辑、一个歌手——并允许客户端获取这些附加的信息。如表 4-3 所示，不是所有的方法都能支持每一个资源，有些情况没有办法执行某些特定的操作。表 4-3 所示为如何在用户播放列表中管理一首歌。

表 4-3 播放列表成员资源可用的 HTTP 方法

示例 URL: `http://example.org/playlists/324/my-favs/41121`

HTTP 方法	结果行为
GET	获取播放列表条目的元信息 (可能是作者、流 URL、长度、风格)
PUT	如果不存在，添加歌曲 41121 到“my-favs”列表
POST	不支持
DELETE	从播放列表删除歌曲 41121

REST 服务并不一定要用 JSON，不过这事实上已经是 Web 的标准，JSON 由于其简单、结构紧凑、比 XML 易读等特点而广为流行。代码清单 4-1 所示为使用 GET 方法请求一个播放列表条目时返回的 Web 服务响应的例子。

代码清单 4-1 GET `http://example.org/playlists/324/my-favs/678632` 的响应

```
{
  "SongID": "678632",
  "Name": "James Arthur - Young",
  "AlbumURL": "http://example.org/albums/53944",
  "AlbumName": "Unknown",
  "ArtistURL": "http://example.org/artists/1176",
  "ArtistName": "James Arthur",
  "Duration": "165"
}
```

如果你想比较 REST 和 SOAP，那么有几件重要的事项需要指出。首先，由于只有 4 个 HTTP 方法可以使用，REST Web 服务的结构通常是可以预测的，所以开发起来比较容易。在你看过几个 REST 服务之后，再学习使用新的 REST API 就简单容易多了。如果和 SOAP 服务开发比较的话，你会发现每一个 SOAP 服务都使用不同的约定、标准，以及 ws-* 规范，集成困难有多大可想而知。

从 Web 服务发布者的角度看, REST 也比 SOAP 更轻量, 因为 REST 只需要创建一个在线 wiki 就可以了, 在 wiki 里定义各种资源, 每种资源的 HTTP 方法, 以及请求响应的例子以展示数据格式。你可以用 Web 技术开发一个可以实用的 REST 资源, REST 框架 (或者容器) 无须提供什么特别的功能, 仅仅就是 HTTP 服务器的基本功能, 有一个把 URL 模式映射到代码的路由机制就可以了。REST 代替 SOAP 的另外一个好处是无须再去管理 WSDL 及 XSD 这种超级复杂的 API 约定。

从客户端视角看, 集成 REST 服务有优点也有缺点。客户端没法自动生成客户端代码及没法发现 Web 服务, 这是缺点。同时, REST 服务也没那么严格, 服务端发布变更的时候, 无须重新编译部署客户端。另一个常见的问题是服务发现, 服务提供者构建发布常用语言的客户端代码库。这种方式下, 客户端代码只需要写一次就可以被复用给不同的客户。显然, 这种方式把更多的负担转移到服务提供者一端, 降低了使用者的难度, 相对自动生成代码, 提供的抽象层次也更好一点。

从安全的角度看, REST 服务没有 SOAP 服务控制的那么精细。为了允许授权访问 REST 资源, Web 服务通常需要在访问 API 之前进行权限验证。客户端首先需要进行认证 (通常使用 OAuth 2), 然后在每次请求的请求头都带上认证令牌。REST 服务也可以依赖 HTTPS 提供传输层安全保障, 无须自己实现消息加密机制。这些折中使得 REST 对于跨 Web 平台开发更简单, 也使得进行企业级集成更困难, 特别是当你需要一些高级特性的时候, 比如要实现语义上的严格一次提交。

从伸缩性角度看, REST Web 服务的一个重要优点有点像本节前面讨论过的那个例子, 它是无状态的, 所有使用 GET 方法的公开操作都可以透明地通过 HTTP 实现缓存。路由一个 REST 请求唯一需要的就是 URL, 所以可以在客户端和 Web 服务之间通过 HTTP 缓存实现 GET 请求的缓存。这样热门资源的大部分访问流量都可以通过反向代理返回, 极大地减轻了 Web 服务及数据存储的负载压力。

你可能已经看出来了, REST 并不一定就比 SOAP 好, 并不能取代 SOAP——仅仅是 SOAP 的一个代替。从大企业角度看, REST 并不成熟、不严格特性也不丰富。从创业公司角度看, SOAP 又太过笨重、严格、难以驾驭。这真的要看你具体应用的细节和系统集成的需求。就像曾经说过的, 如果你只是需要暴露 Web 服务给移动客户端及一些第三方网站, REST 就是一种不错的选择, 如果你是一家创业 Web 公司, REST 也更容易起步,

更容易和现有 Web 技术集成，无论客户端使用什么技术开发都可以。

我们已经讨论过 Web 服务的类型和不同的设计方式，下面让我们花一点时间看一下如何实现 Web 服务的伸缩。

伸缩 REST Web 服务

要实现 Web 服务层的伸缩性，主要用到第 2 章提到的两个技术：把 Web 服务层按功能切分成比较小的粒度，然后增加备份实现伸缩。首先要做到良好设计的 Web 服务，才能使用这两种技术进行伸缩。

保持服务无状态

类似于应用的前端层，需要小心地处理 Web 服务的状态。绝大多数可伸缩方案都要求 Web 服务本身是无状态的。这就意味着需要把 Web 服务所有的共享状态移出到共享数据存储设备上，比如对象缓存、数据库、消息队列等。保持 Web 服务无状态会带来很多好处。

- 可以基于每个请求进行流量分派，将负载均匀分布到多台 Web 服务节点上。在 Web 服务与客户端之间部署负载均衡器，每个请求都可以发送到任意一个 Web 服务节点上。通过轮询的方式分发请求可以获得更好的负载分派及更好的弹性。
- 由于任意 Web 服务请求都可以被任意 Web 服务节点处理，所以当 Web 服务节点宕机的时候，可以将其从负载均衡池中剔除。大多数的负载均衡都支持心跳检测，确保 Web 服务节点可以有效处理请求。一旦某个节点宕机，或者出现某种失效，负载均衡器就会将该节点从负载均衡池中移除，这样虽然整个 Web 服务集群的处理能力降低，但是客户端不会因为响应超时而出现故障。
- 使用无状态的 Web 服务，可以在任何时候重启或者下线 Web 服务节点，而无须担心对客户端造成任何不良影响。例如，如果你想关闭某台服务器进行维护，你就把这台服务器从负载均衡池中移出去好了。大多数负载均衡器都支持优雅移除服务器，客户端新来的连接请求不会再连接到已移除的服务器上，但是已建立的连接也不会中断，以保证不会出现客户端异常。从负载均衡池中移除服务器后，还需要稍等一会，等所有客户端连接都断开，这可能需要一两分钟的时间，然后就

可以放心地关闭服务器了，不用担心会影响任何一个请求。

- 和服务器下线类似，还可以实现 Web 服务更新的高可用。可以以轮流的方式将一台台服务器从负载均衡池移出、更新，然后重新放回负载均衡池。如果你的软件不允许同时部署两个版本，那么一个办法就是分别部署两个版本，然后在负载均衡层面上将访问从一个版本切换到另一个版本。不管你选择何种方式，无状态对于 Web 服务都意味着易于维护。
- 通过将所有的应用状态从 Web 服务中移除，Web 服务可以通过增加备份实现伸缩。要做的仅仅是添加机器到负载均衡池增强集群的并发处理能力。这里假设你的数据持久层需要进行垂直伸缩，我们在第 5 章会探讨这部分的内容。
- 如果使用亚马逊弹性负载均衡器或者 Azure 负载均衡器（这种支持自动伸缩功能的云服务），那么就跟前端一样，可以实现 Web 服务的自动伸缩。任何时候，当一台服务器宕机，负载均衡器就会立即启动一个新实例代替，如果服务器非常繁忙，也会增加一个新的实例提供更多负载处理能力。

如你所见，保持 Web 服务无状态会对系统的伸缩性和可用性带来很多好处。只有一类状态储存在 Web 服务节点上是安全的，那就是无须同步也无须担心不可用的缓存对象。根据缓存的定义，缓存数据是可丢失的，也可以在任何时候重新构建，所以服务器宕机不会引起数据丢失。在第 6 章我们会探讨更多关于缓存的内容。任何需要跨 Web 服务保持数据一致的方案都会导致响应延迟或者系统不可用故障。要确保系统不会惹上这样的麻烦，最稳妥的办法就是在 Web 服务中只存储缓存对象，而且这些缓存只基于绝对时间进行过期失效。这样对象就可以分别存储在各个 Web 服务上直到失效，而无须各个 Web 服务之间互相通信。

如果 Web 服务需要存储用户状态，那么最好找一些持久存储的替代方案。图 4-5 所示为无状态的 Web 服务如何通过外部数据存储、缓存、消息队列实现数据持久存储与访问。每个数据存储组件都表示一个状态持久化的地方，每个数据存储组件都可以使用不同的技术针对具体场景采用合适的方式实现，也可以用同一个存储完成。

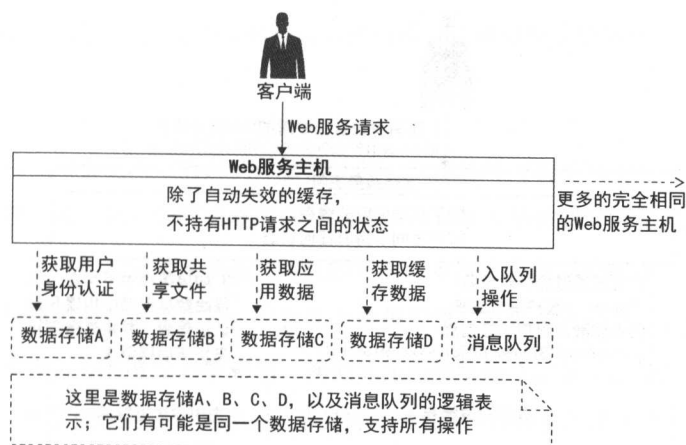


图 4-5 应用状态从 Web 服务主机中剥离出来

在构建无状态 Web 服务的时候，会遇到一些常见的场景，要求不同 Web 服务之间共享状态。

第一种场景和安全有关，Web 服务需要客户端在每次 Web 服务请求的时候都传递认证令牌。令牌必须在服务端进行校验，确保用户有权限执行他调用的操作。可以在 Web 服务节点上直接缓存身份认证及权限校验结果，但是当权限变更的时候或者账户被禁用可能会带来问题，所以这些缓存对象必须新的权限生效之前失效。如果缓存对象使用全局缓存，只有一个备份，那么当用户权限变更的时候，失效操作就会比较容易。图 4-6 所示为如何从共享对象缓存中获取权限认证信息。第 6 章将进一步讨论对象缓存的具体细节，现在，仅仅了解缓存可以将 key（比如认证令牌）和对象（比如序列化的权限信息）映射起来就好。

另一种常见的情况是无状态的 Web 服务如何处理资源锁。如第 3 章中提及的，可以使用分布式锁系统比如 Zookeeper，也可以使用数据存储开发一个简单的锁服务。不过为了 Web 服务具有良好的可伸缩性，最好避免使用资源锁，寻找其他更好的同步并发进程的方式。

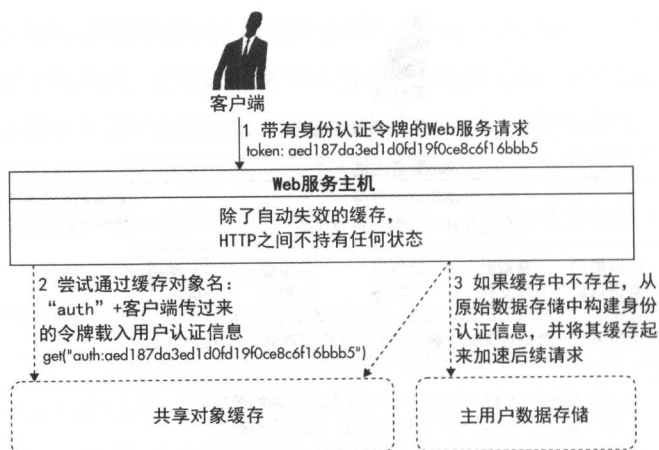


图 4-6 从共享对象缓存获取身份认证信息

分布式锁很有挑战，每个锁都需要一个远程调用，决定服务阻塞还是失败。反过来也会增加响应延迟及服务的并发度。除了资源锁，还可以使用乐观并发控制，这种方案无须获取锁，只需要在最终更新之前检查状态就可以了。也可以考虑使用消息队列实现组件之间解耦合，从根本上去除对资源锁的依赖（我们将会在第 7 章详细讨论消息队列与异步处理）。

提示

如果你决定使用多个锁，应该以特定的顺序获取每个锁，避免出现死锁。举个例子，如果你要对两个用户账号加锁，以便在它们之间实现转账操作，那么每次加锁的顺序必须要一样，比如按照字母顺序，排在前面的账号先加锁。通过这种简单的约束，可以避免死锁的发生，提高服务的可用性。

如果上面提到的这些技术都不管用，你仍然需要使用资源锁，那么你还要做出另一种权衡：是获得多个细粒度的锁，还是获得少量粗粒度锁但是会阻塞大量的数据访问。当你获取多个细粒度锁的时候，由于每个锁都要请求分布式锁服务，调用延迟会增大。获取多个细粒度锁的时候，复杂性的风险也会增大，如何获取每个锁，从哪里获取等这些问题会使得事情变得不那么容易。在代码的不同部分获取不同的锁增加死锁的可能。另一方面，如果你只使用少量粗粒度的锁，调用延迟会降低，死锁风险也会降低，不过同时系统并发度也会下降，多个 Web 服务线程可能阻塞在同一个资源锁上。到底该用细粒度锁还是粗

粒度锁，没有明确的规则，必须具体情况具体分析，工程师要头脑清楚。

提示

伸缩性与资源高效利用的关键是让每台机器都尽可能的独立运行，每台机器都能创建进程（执行计算或者服务请求），尽可能少地依赖其他机器。锁明显是违反这个原则的，它需要机器之间互相通信或者和一个外部系统通信。使用了锁，所有的机器都变得互相依赖。一旦某个进程变慢，其他进程要等待它释放锁，也都跟着变慢。如果某个功能崩溃，其他功能也可能崩溃。你可以在批处理作业中使用锁，在队列任务中使用锁，但是最好不要在 Web 服务的请求响应过程中使用锁，以避免可用性问题，并增加系统并发处理能力。

构建可伸缩无状态的 Web 服务面临的最后一个挑战是应用级别的事务。事务很难实现，特别是当你想在 Web 服务约定层面实现事务保证的时候，需要在多个服务之间实现分布式事务。

分布式事务是指多个内部服务步骤与外部服务调用构成一个整体，要么一起完成，要么整体失败。有点像数据库事务，目标也是一样——所有的变化统一生效，保证从外部看是完整一致的，或者所有的修改都回滚到最初状态，好像什么都没发生过一样。分布式事务作为一个技术难题已经由来已久，简单说就是在不牺牲可用性的前提下，分布式事务很难伸缩整合。实现分布式事务最常用的方法是两阶段提交（2PC）算法。

举个分布式事务的例子：一个为在线商店创建订单的 Web 服务。图 4-7 所示为分布式事务如何执行。在这个例子中，OrderService 依赖 PaymentService 和 FulfillmentService，这两个服务任何一个失败都会导致 OrderService 无效。另外，在整个事务过程中，所有的协作服务都需要保持持久连接及应用资源，以允许当有组件拒绝提交事务的时候整个事务能够回滚。

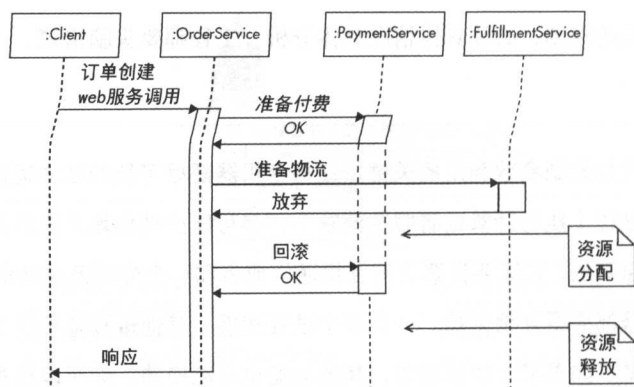


图 4-7 分布式事务失败

对伸缩性和可用性而言，分布式事务使用两阶段提交可谓糟糕透顶。当事务包含的服务越多，依赖的资源越多，两阶段提交的执行难度也越大。另外，失败的概率也越大。这里分享一个我个人的经验，那就是不要使用分布式服务，找一些其他的方法。

取代分布式事务的第一种方法就是：完全不支持分布式事务。这似乎听起来很傻，但是对大多数创业公司而言，权衡开发速度、可用性、伸缩性各方面的得失考量，其实是可行的。例如，对于一个社交媒体网站，如果关注某个人的活动，而这些活动有一部分不能立即反映在搜索索引结构中，当用户搜索关注人活动的时候，就会有一些活动搜索不到。系统的核心功能不能不做，系统又要伸缩性，又要易维护，还要节约开发时间降低开发成本，那么牺牲数据一致性是对企业最有利的办法。

取代分布式事务的第二种方式是：补偿事务机制。补偿事务是指当一个比较大的逻辑事务部分失败的时候，对某些操作的结果进行恢复。回到在线商店这个例子，OrderService 先后发起两个请求，一个是 PaymentService，一个是 FulfillmentService。每个请求都是独立的（无须事务支持）。如果两个请求都成功了，那么什么也不需要做，事务就是成功了。如果 PaymentService 失败了，OrderService 终止就可以了，不会向 FulfillmentService 发送请求。如果 PaymentService 成功了，而 FulfillmentService 失败了，那么 OrderService 还需要再请求一次 PaymentService，将前面支付的款项退还回去。图 4-8 所示为这种乐观控制方式的具体执行时序图。

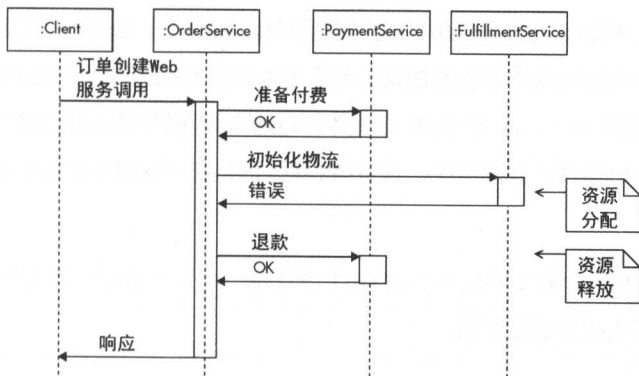


图 4-8 补偿事务以修正部分执行

这种方式的好处是 Web 服务不需要互相等待；事务执行期间也不需要维护任何状态和资源。每个服务应答都是一次彼此隔离的调用。协作服务（这里是 OrderService）只需要对每次 Web 服务调用的数据一致性负责就可以了。另外，补偿式事务通常以异步方式处理，把补偿消息丢到消息队列里去，而不阻塞客户端调用。

在各种情况下，我最先问的问题是事务甚至锁是否一定是必要的。对于一个创业公司，这类复杂性会极大地增加开发和维护的代价。如果你能把这些东西简化，比如你可以使应用优雅地处理失败，而不是不惜代价地保证成功，通常是一种更好的选择。你也应该尽可能地使用数据存储完成事务支持。大多数数据存储都或多或少地支持原子操作，原子操作可以用来实现简单的事务或者排他性资源访问。我们将在第 5 章详细讨论数据存储与事务支持。

缓存服务响应

利用 HTTP 协议缓存是可伸缩的 REST Web 服务的另一项重要技术。对 Web 应用而言，HTTP 缓存是一个强有力的伸缩性技术。同样，基于相同的知识、技能和技术，HTTP 缓存对 Web 服务的伸缩性也很有价值。第 6 章将详细讨论 HTTP 缓存，这里我们快速过一下如何利用 HTTP 缓存构建 REST Web 服务。

我前面提到，REST 服务会用到所有的 HTTP 方法（比如 GET 和 POST），如果正确使用，需要遵守每一种方法的语义约束。从缓存角度看，GET 方法是最重要的一个，因为 GET 响应是可以被缓存的。

HTTP 协议要求所有的 GET 方法调用都是只读的。如果一个 Web 服务请求是只读的,那么请求不会带来任何改变。也就是说,无论是否发送 GET 请求给 Web 服务,都不会引起 Web 服务状态的改变。既然 GET 请求是否到达 Web 服务没有什么区别,那么就可以利用代理或者客户端自己缓存响应,通过缓存返回响应,从而跳过 Web 服务,减轻 Web 服务的压力。

对于 HTTP 缓存,需要确保所有的 GET 方法都一定是只读的。请求任何 GET 资源都不会引起状态变化或者数据更新。

Web 应用打破 GET 方法只读特性最让人吐槽无力的用法就是在 GET 方法中改变状态。在十几年前,经常会看到 Web 应用在 GET 请求中对数据库进行更新。例如,通过 GET 请求(URL: <http://example.com/subscribe?email=artur@ejsmont.org>) 订阅一个邮件列表。虽然这样对于开发者比较方便,但是显然会引起应用状态的改变,发送请求还是不发送请求结果大不相同。

现在很少会看到有 REST Web 服务以如此明目张胆的方式打破这一规则。不幸的是,它还是会以其他方式出现,在不经意间就陷入麻烦。例如,在我曾经工作过的某个公司,我们就不能在 Web 应用前端使用 HTTP 缓存,因为商业智能和广告团队需要 Web 服务器的日志生成报告及计算收入划分。这就意味着,即使我们的 Web 应用正确地实现了 GET 方法,所有的 GET 方法都是只读的,我们也不能在 Web 集群的前端加一个缓存代理层,因为这样 Web 服务器会失去大量的访问请求,进而减少日志收集导致报告偏差。

另一个会打破 GET 请求语义规则的方式是在 Web 服务节点上使用本地对象缓存。例如,在一个电子商务 Web 应用中,需要调用一个 Web 服务获取某个商品的明细。客户端要发送一个 GET 请求获取数据,这个请求会通过负载均衡路由到某个 Web 服务节点。这个节点从数据存储上加载数据,并把结果记录在本地缓存,然后返回响应到客户端。如果商品明细在放入缓存后就被更新了,那么另一个 Web 服务节点生成并放入缓存的响应数据就会和这个节点不同。虽然两次 GET 操作都是只读的,但还是对 Web 服务集群的行为产生了影响。现在,每一次 GET 请求连接到不同的服务节点上得到的响应结果是不同的,有的是新数据,有的是老数据,在客户端看来就会比较诡异。

另一个需要考虑的重要方面是身份认证,设计 REST API 的时候会遇到有的 API 需要认证,有的不需要。REST 服务通常通过请求头传递认证信息,Web 服务通过验证请求头

决定是否提供访问服务。问题是不同用户的权限不同，验证过的 REST 终端可以看到的数
据也会不同。这就意味着针对特定用户，仅仅依靠 URL 不足以产生响应结果。相应地，
HTTP 缓存在构建缓存 key 的时候需要包含认证头。如果不同用户看到的数据是不同的，
那么这种缓存分隔的方法就还不错；如果看到的数据是相同的，就会造成浪费。

提示

可以通过利用 HTTP 头实现被认证的 REST 资源缓存。认证信息包含在 Web
服务响应中，响应通过头信息指导 HTTP 实现响应的分隔缓存：不同认证头的响
应对应的缓存也不同（每个用户的缓存都被分隔开来）。

为了尽可能地利用 HTTP 缓存，需要尽可能地使资源可公开访问。公开的资源只需要
为每个 URL 提供一个缓存，极大地提高缓存的效率，降低 Web 服务的负载。

举个例子，如果你想开发一个社交音乐网站（比如 www.grooveshark.com），用户可
以听音乐，可以分享播放列表，这种情况大部分 GET 方法都可以是公开的。你需要限制
用户访问专辑明细、歌曲、歌手，甚至播放列表吗？可能不需要。使 GET 方法公开，你
可以在缓存层忽略用户信息，更高效地利用缓存对象。

在创业公司开发的早期，你的 Web 服务层也许并不需要 HTTP 缓存，但还是值得去
思考这方面的内容。HTTP 缓存在 Web 服务层的实现方法和在前端层的实现方法差不多。
为了缓存的伸缩性，通常在客户端和 Web 服务之间部署一个反向代理。意味着有些事情
是依赖于 Web 服务组织，以及如何被使用。图 4-9 所示为带有反向代理的 Web 服务的常
见部署方法，反向代理部署在 Web 服务和前端应用之间。

随着 Web 服务层的发展，部署方法可能会变得更复杂，每个 Web 服务都有一个自己
的反向代理专门为自己提供缓存。根据反向代理的用法，也许还需要在反向代理和 Web
服务之间部署一个负载均衡器，进行流量分发和提供快速失效恢复。图 4-10 所示为这种
部署方式。

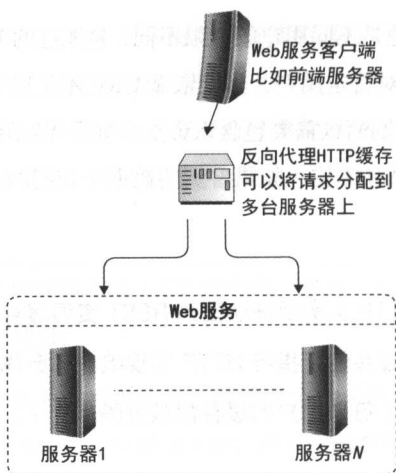


图 4-9 客户端和服务之间的反向代理

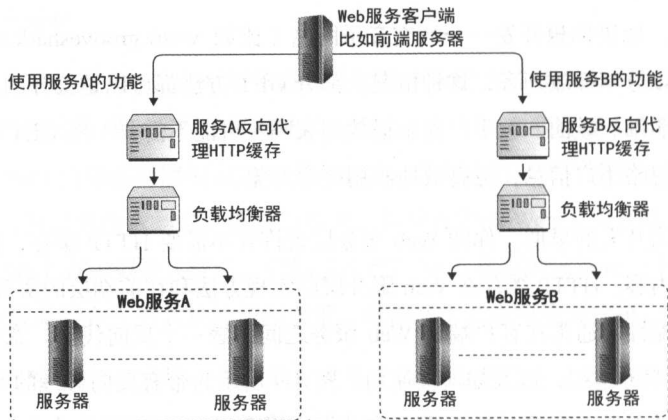


图 4-10 反向代理在每个 Web 服务的前端

这种配置的好处是每个请求都会经过反向代理，无论请求从哪里来。随着 Web 服务层的发展，系统逐渐向面向服务架构演化，这些考量会让你获益良多。独立对待每一个 Web 服务，以相同的方式对待所有的客户端，无论它们在哪个层次，是否降低耦合提高抽象级别，都一视同仁。现在，让我们探讨一下 Web 服务独立性与隔离性对伸缩性的影响。

功能分割

在第 2 章的时候我们提到过功能分割是伸缩性的关键技术之一。究其本质，功能分割

就是把一个大系统切分成一大堆小的、低耦合的小部件的方式，相对于大系统需要运行在一个超强计算能力的服务器上，这些分割后的小部件可以运行在更多的普通服务器上。在不同场景中，功能分割代表的意思可能不尽相同。对于 Web 服务而言，功能分割指的就是把一个服务切分成一堆小的、相对独立的 Web 服务，每个小的 Web 服务都只关注整个系统功能的一小部分。

为了更好地理解，我们举个例子。假设你要开发一个电子商务网站，你可以在一个 Web 服务中实现所有的功能，这个 Web 服务处理所有的请求，如图 4-11 所示。

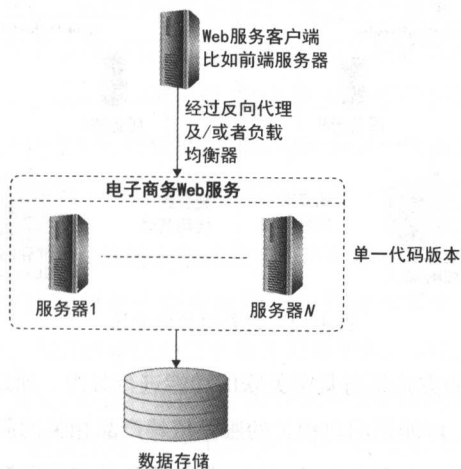


图 4-11 单一服务

还有一种方法，你可以把系统切分成更小，更低耦合的多个 Web 服务，每个 Web 服务都只关注一个比较窄的职责范围。例如，可以把产品类目相关的功能提取出来，构建一个独立的 Web 服务叫 ProductCatalogService。这个服务可以创建、管理、搜索产品及产品描述、产品价格，以及产品分类。类似地，还可以抽取用户相关的功能创建一个 UserProfileService，管理账户，更新信用卡信息，打印历史账单明细。

相比于单个巨大的紧耦合的 Web 服务，两个更小、更独立、更内聚的 Web 服务更有优势，可以在基础设施、数据库、开发团队各方面获得更松散的耦合性。概括来说，功能分割就是观察一个系统，将那些紧密联系的子功能识别出来，将这个子功能提取出来构成一个独立的子系统。

图 4-12 所示为功能分割的 Web 服务架构图。功能分割的好处就是有了两个独立的子系统，可以使用至少两倍的硬件服务器，这一点对于数据存储层特别有帮助，尤其是使用传统的关系数据库的时候，因为关系数据库很难进行伸缩。

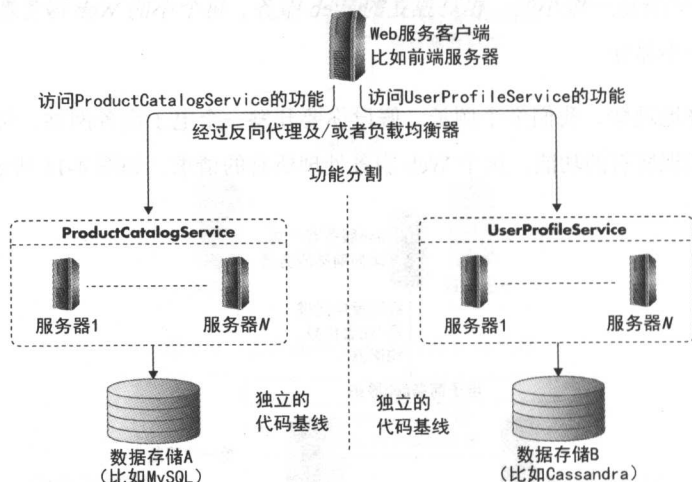


图 4-12 服务的功能分区

由于实现功能分割的方式是对紧密关联的功能进行分组，所以分割后新的 Web 服务之间会有一些依赖关系。可能是用户相关的服务依赖产品相关的服务，或者是产品相关服务要引用用户服务的数据，但是大多数时候，服务的开发和变更都是独立的，影响仅限于服务内部。这对团队的工作也有好处，工程师不需要了解整个系统就可以完成需求变更，团队可以负责开发一个或者几个 Web 服务，而不会动整个代码库。

功能分割的另一个重要影响是每个 Web 服务都可以独立伸缩。我们先看下 ProductCatalogService，这个服务收到的请求应该主要是一些读请求，而不是数据更新请求，每次搜索，每次页面浏览都需要载入各种产品数据。而对于 UserProfileService，访问模式可能完全不同。用户只想访问自己的数据（这个特点有助于构建索引及易于实现分布式数据存储），当你需要记录用户浏览了哪些产品的时候，需要执行的也主要是写操作。对于一个在线商店而言，数据集可能会非常庞大，用户数量的增长速度常常远高于产品数量的增长速度。

这些访问模式的不同会导致伸缩性需求也大不相同，不同服务的设计约束也不一样。这两种服务使用相同的缓存是否合理？使用相同类型的数据存储是否合理？这两个服务对于业务而言是否同样重要？存储的数据特性是否一样？是否需要使用相同的技术栈实现这些有着很大不同的服务？如果这些问题的答案都是“否”，那真是最好不过了。通过将 these 服务分割开来，可以得到更大的选择余地，可以针对性地选择更合适的工具，可以根据每一种 Web 服务选择对应的伸缩性策略，而不是逼着自己只能用一种方案在各种需求之间左支右绌。

在小型创业公司早期，上面提到的很多东西可能并不是很有必要，但是你要考虑到你的系统会发展会增长会扩大会更复杂，你会逐渐对 Web 服务层进行功能分割，系统逐渐变成面向服务的架构，服务会成为一等公民，服务职责会越来越单一，封装性会越来越好，耦合性会越来越低。不止是在类或者组件层面运用前面提到的各种设计原则，还会在 Web 服务层面运用这些原则，在整个系统层面实现弹性、复用、易维护这些特性。

功能分割太早、划分粒度太细的主要挑战在于新的需求不断出现，需要合并多个 Web 服务的数据和功能。回到我们电子商务的例子，假设你需要开发一个新的推荐服务 `RecommendationService`，你应该意识到这个服务需要依赖产品目录数据和用户信息数据，以构建用户相关的推荐模型。在这种情况下，推荐服务需要以独立实体的方式集成另外两个服务，这要比访问一个数据存储一个代码库的开发工作量更大。不过，尽管服务集成带来很多挑战，但是功能分割仍然是一项非常重要的伸缩性技术。

小结

良好设计与良好结构的 Web 服务可以在很多方面带来助益，在长期维护的成本方面，在系统的局部简化方面，会对伸缩性带来一些积极影响，但是如果说有 Web 服务层一定会带来好处，或者说对任何应用都带来好处，这话就有点不负责任了。初创公司要在很多不确定性及很大时间压力下开展工作，所以需要更加小心行事，不要过度设计，不要浪费宝贵的时间做过多的前瞻性开发。如果你需要集成第三方服务，或者支持移动客户端，那么可以从一开始就把这些需求当作 Web 服务来开发。只有在技术团队成长壮大，发展成两三个敏捷团队（二十多个工程师）的时候，面向服务的架构和 Web 服务的价值才开始体现出来。

我建议你进一步学习关于 Web 服务^[46, 51]，构建 REST 与 Web 服务的现代方法^[20]，以及 SOAP^[31]与面向服务的架构。

构建可伸缩的 Web 服务有一种相对简单的方法，就是把 Web 服务节点的状态全部移出去，并使用激进的缓存策略。我确信你已经思考过“我要在哪里存储这个状态”或者“我们如何才能等量地伸缩这些状态组件，保持和前端与 Web 服务一样的处理能力与规模”。这些都是一些很不错的问题，当我们尝试着去回答这些问题的时候，我们就已经触及到伸缩性问题最有挑战最激动人心的部分：数据存储层的伸缩性。

5

数据存储层

传统公司习惯上对数据库进行垂直伸缩，他们购买更强大的服务器，添加更多内存，安装更多磁盘，期望数据库引擎能够利用这些资源实现伸缩。绝大多数传统公司都是这样做的，只有最大的最成功的公司才需要进行水平伸缩。随着互联网和社交网络的崛起，以及软件工业的全球化，一切都改变了，系统需要支撑的数据量和并发用户量如火箭一样，直线上升。今天，百万级的用户和十亿级的数据量都非常常见，软件工程师需要更好地了解数据存储伸缩性有关的技术和工具。

前面几章中，我们伸缩前端和 Web 服务层的手段是将状态从服务器上移出，保持服务器无状态，当我们需要伸缩的时候，只需要增加机器就可以了。现在我们讨论保存状态的数据层的伸缩性问题，如何实现水平伸缩，如何不成为系统的瓶颈。

根据具体业务需求，可能需要对应用和数据模型进行伸缩，你可以使用传统的关系数据库引擎，比如 MySQL，或者是比较前沿的非关系型数据存储。这两种存储各有优缺点，本章会尽量客观地介绍这两种方案如何针对不同应用场景进行互相补充。我们先以 MySQL 数据库为例讨论关系数据库引擎的伸缩性。

MySQL 伸缩性

MySQL 是目前最流行的数据库，而且还会继续流行相当长一段时间。主流关系数据库有十几种，MySQL 的性能和伸缩性更适合大多数互联网创业公司的需求。伸缩 MySQL 比较困难，从第一天开始就要好好规划，不过世界上最大的一批互联网公司都成功使用 MySQL 实现了伸缩性，包括 Facebook^[L35]、Tumblr^[L33]，以及 Pinterest^[L31]。现在我们先看看伸缩 MySQL 最主要的手段之一：复制。

复制

复制通常指的是在多个机器节点上存储同一份数据的多个备份。不同数据存储系统实现复制的方法不同。对于 MySQL 而言，复制就是在两台服务器之间同步数据，其中一台服务器叫做主节点，另一台服务器叫做从节点。本章中，我们也会讨论如何在更多的 MySQL 服务器之间同步数据，不过关注的重点还是主从复制。

使用 MySQL 复制的时候，应用可以通过从服务器读取数据，但是写数据只能通过主服务器。所有的数据修改命令，诸如 update、insert、delete 或者 create table，只能发送给主服务器。主服务器将这些更新操作记录到一个叫作 binlog 的日志文件中，每个操作都附带一个时间戳，并拥有一个序列号。一旦操作写入 binlog，就可以发送到从节点。

图 5-1 所示为 MySQL 数据复制的工作原理。首先，客户端连接到主服务器，然后执行一个数据更新操作。更新操作被执行并写入 binlog 文件中。此阶段，主服务器会返回一个响应给客户端并继续处理其他事务。从服务器可以在任何时间点连接主服务器，请求主服务器 binlog 文件的增量更新部分。请求中，从服务器提供本机拥有的最新命令序列号。由于 binlog 文件的所有命令都按序列号存储，主服务器可以迅速定位到正确的位置将新增部分以数据流的方式返回给从服务器。从服务器将这些数据流写入自己的和主服务器 binlog 文件相对应的 relay 日志文件中，然后执行这些操作，增量部分的数据就同步到从服务器上了。

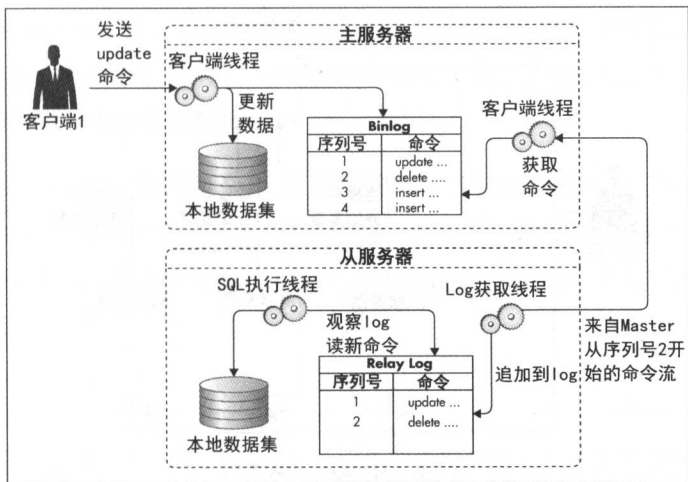


图 5-1 MySQL 复制

关于 MySQL 复制需要注意的一件重要的事是异步性，意思是主服务器不会等待从服务器的复制操作。无论从服务器是否连接，主服务器都会把命令写入 binlog。从服务器需要知道自己的数据复制到哪里了，进而确保更新操作正确，主服务器则无须关心从服务器的任何情况。从服务器与主服务器断开连接，主服务器无视之即可。MySQL 复制的异步性将主从服务器的耦合关系分解开来，可以在任何时间连接一个新的从服务器，也可以在任何时间断开一个从服务器，都不会影响到主服务器。

由于复制是异步的，主服务器无须记录从服务器的状态，因此复制策略可以很灵活。例如，可以配置多台从服务器，实现分布式的查询操作。实践中，通常每个主服务器会配置两个或者更多个从服务器。

图 5-2 所示为一主多从的配置方案。每个从服务器都记录自己复制的最后一命令，都连接到主服务器等待新的数据到达，但是从服务器之间并不会有任何交互。任何从服务器都可以在任何时间点与主服务器建立连接或者断开连接，不会影响其他从服务器。

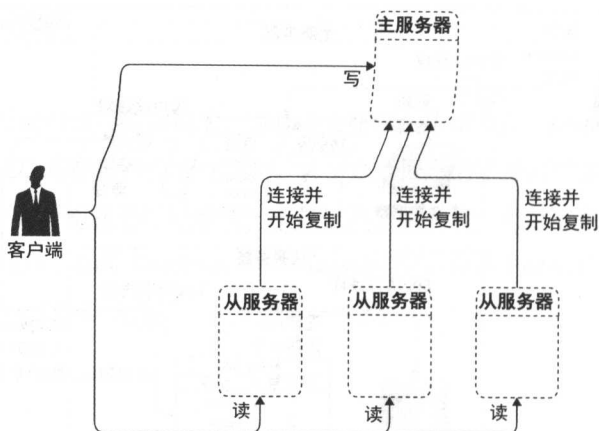


图 5-2 MySQL 多从复制

多个从服务器可以带来很多的好处。

- 可以将只读操作分布在多个从服务器上，从而将负载分摊到多台机器上。这种伸缩性方法是增加备份（第 2 章解释过）在数据库引擎上的应用，可以为同一份数据添加更多的备份以提高读操作能力。
- 可以针对不同类型的查询使用不同的从服务器。例如，一台从服务器专门用于一般的应用查询，而另一台服务器专门用于耗时较长、执行很慢的报表类查询。通过将慢查询放在一台单独的服务器上，将应用与那些强 I/O 的查询隔离开来，进而改善用户体验。
- 利用 MySQL 复制的异步特性可以实现一个零停机时间的备份。执行一个从服务器的一致性备份非常容易——只需要关闭 MySQL 进程，将数据文件复制到存档目录，然后重新启动 MySQL 即可。MySQL 重新启动后，会连接到主服务器上，将这段时间丢失的更新操作复制到本机。
- 如果有一台从服务器宕机，只需要不再发请求给这台服务器（从轮询请求中剔除出去）即可，直到这台服务器恢复。失去一台从服务器也不是什么大事，不会将失效信息传递给其他机器。MySQL 服务器不会记录其他服务器的可用性，所以检测服务器失效必须在客户端进行。可以在应用逻辑中实现失效检测功能，也可以使用一个智能的代理或者负载均衡器完成失效检测。

人们使用 MySQL 及其他数据存储复制的一个主要原因是增加可用性,当出现数据库宕机的时候,减少替换的等待时间。如果使用 MySQL 复制,需要面对的失效场景有两种:从服务器失效,以及主服务器失效。

从服务器失效一般说来问题不大,可以很快修复,只是在宕机期间不能向失效的从服务器发送查询请求。数据库的处理能力有所下降,但是系统的可用性几乎不受影响,只要把失效的服务器从访问轮询中剔除就可以了。当失效的服务器重建完成,再加回到集群中,系统恢复正常。

提示

有一点需要注意的是,MySQL 从服务器的重建是一个手工的过程,需要通过主服务器或者其他从服务器完成一个数据库完全备份操作。MySQL 不支持从一个空数据库上启动一个从服务器。要想启动一个从服务器并继续从主服务器上复制更新,需要做一次数据全量备份,并得到最后更新的操作序列号。然后就可以启动从服务器,从服务器会获取积压的数据。全量备份的数据越旧,数据库越忙,新启动的从服务器就越需要更长的时间去赶上最新的数据更新。对于一个比较忙的数据库,可能需要花费几个小时才完成从服务器的数据更新,然后才能把这台从服务器加回到访问轮询中。

重建从服务器看起来很麻烦,但是和主服务器失效恢复比起来还是轻松多了。MySQL 不支持自动失效转移,也不支持自动将一台从服务器提升为主服务器。一旦主服务器挂掉,就会有一大堆事情需要做。首先,要找出哪个从服务器的数据是最新的(哪个从服务器有最大的更新操作序列号)。然后重新配置,将这台从服务器配置为主服务器。如果有多个从服务器,需要通过新的主服务器备份重建这些从服务器,或者手工调整所有服务器的 binlog 和 relay 日志文件,确保从服务器和新的主服务器是一样的。最后,需要配置其余所有的从服务器从新的主服务器上复制更新。根据具体配置情况,这个过程可能挺简单,也可能挺复杂,但对于大多数工程师而言,都是一场噩梦。

主服务器失效恢复的麻烦促使人们设计出另一种复制部署结构,就是主-主结构。这种结构中,有两个服务器可以接受写操作,主服务器 A 从主服务器 B 复制数据,同时主

服务器 B 从主服务器 A 复制数据。MySQL 复制策略允许这种环形复制，写入到主服务器 binlog 文件的更新操作包含最初写入时的服务器的名字。通过这种方式，任何发送给服务器 A 的更新操作都会复制到服务器 B，但不会再复制回服务器 A，服务器 A 知道自己已经执行过这个操作了。

主-主部署模式如图 5-3 所示。所有发送给主服务器 A 的写操作都记录在 binlog，主服务器 B 复制这些写操作到自己的 relay 日志并执行，从而获得 A 的数据备份。主服务器 B 也会把这些更新操作写到自己的 binlog 文件中，从而使其他从服务器可以复制这些更新操作。同样，主服务器 A 也会从主服务器 B 的 binlog 文件中复制更新操作，并写入自己的 relay 日志文件中，执行新的更新操作，再写入自己的 binlog 文件。

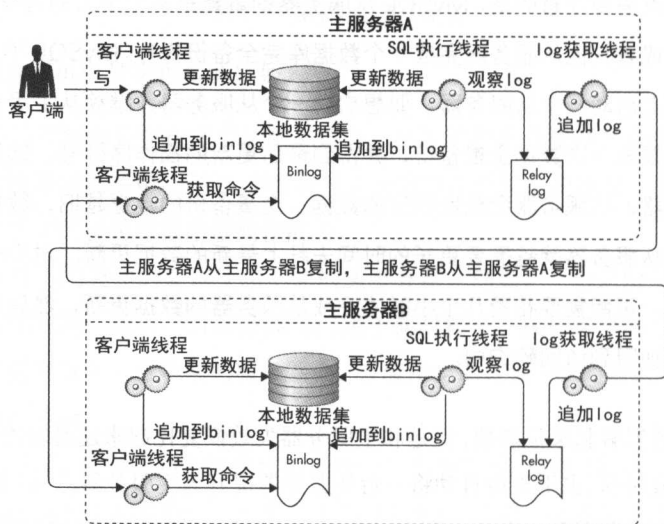


图 5-3 MySQL 主-主复制

这种拓扑结构更复杂，但是对于主服务器失效这种场景，可以实现透明维护及快速恢复。如果主服务器 A 失效，或者要对主服务器 A 执行一个较长时间的停机维护，应用可以快速切换连接到主服务器 B 上。

图 5-4 所示为如何构建主-主复制的模式。主服务器 A 和主服务器 B 的从服务器数量都是一样的，应用不管连接哪个集群，处理能力都是一样的。如果主服务器 A 宕机，应用可以快速切换到主服务器 B 的集群。

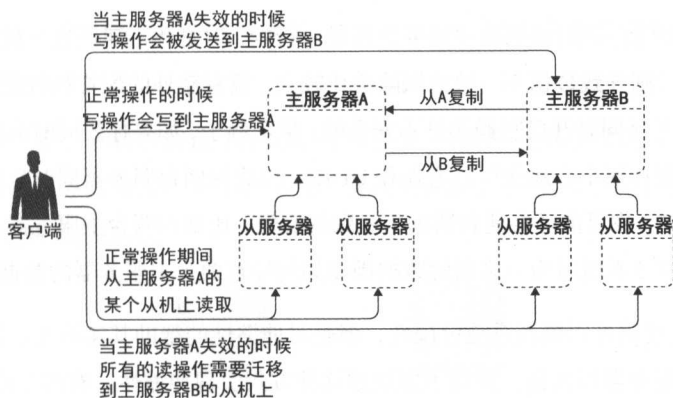


图 5-4 MySQL 主-主失效恢复

有两个对等的主-主架构的服务器集群，应用可以随意进行切换而影响最小。例如，需要升级主数据库服务器的硬件或者软件，这个过程要一个小时，主-主结构可以做到应用的影响降低到只有几秒，但是有个前提，就是一个时间段内只能升级一个服务器集群。首先，升级备用主服务器 B 及其从服务器。然后，关闭对主服务器 A 的写入操作，再等待一段时间，等所有主服务器 A 的数据都同步到主服务器 B。随后就可以将应用所有的写操作都重新配置指向主服务器 B，这时主服务器 B 已经同步了所有主服务器 A 的数据，所以不会出现数据冲突的风险。等到应用都重配置完毕，停机时间就结束了，读写操作恢复正常。最后再在主服务器 A 和从其服务器上执行升级维护。图 5-5 所示为整个维护期间的详细步骤。

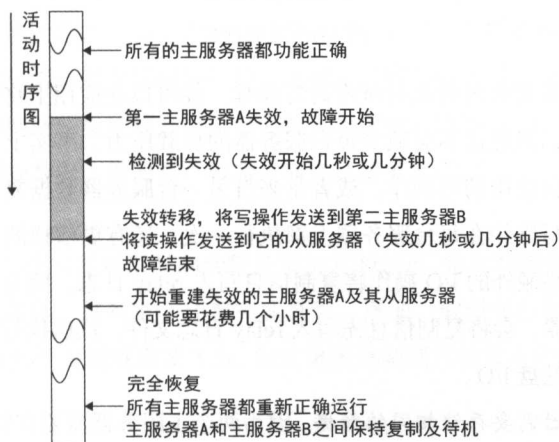


图 5-5 维护失效的时间线

虽然在理论上,同时向两台主服务器写是可行的,但是我不建议这么做,一则是事情会比较复杂,二则导致数据不一致的风险会比较大。没有额外的配置和数据场景分析,只是向两台主服务器同时开启写操作是不安全的。举个例子,如果你想同时向两台主服务器写数据,就需要特别小心地使用自增长和 UUID,以确保两台服务器同时生成的序号不相同。即使如此,你可能还会遇到数据不一致的麻烦,比如在两台主服务器上同时更新了同一个数据。图 5-6 所示为一系列的数据操作如何导致两台主服务器的数据不一致。

虽然主-主复制可以提高系统可用性,但是对伸缩性的帮助其实不大。你越是要避免同时往两个主服务器写数据,就越不能通过这种方式实现伸缩性。有两个原因导致主-主复制不是一种可行的伸缩性技术。

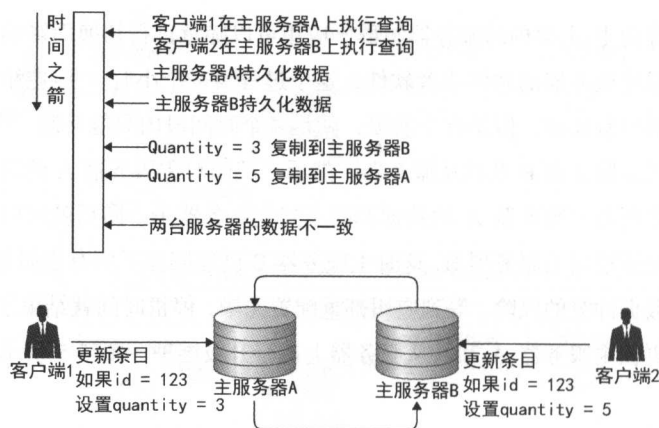


图 5-6 更新冲突

- 两个主服务器需要同时执行所有的写操作。你可以在应用上将写操作分布到两个主服务器上，但是这不会减少每台服务器的负载压力。事实上，每台主服务器都需要执行来自应用的写操作，或者是来自另一台服务器数据复制的写操作，最终就是所有的写操作在每台服务器上都执行一次。还有更糟糕的，每台主服务器都需要执行一些额外的 I/O 操作将复制信息写入 relay 日志。因为每台主服务器也是一个从服务器，会将复制信息先写入 relay 日志文件，然后执行更新操作，自然会引起额外的磁盘 I/O。
- 两个主服务器需要存储相同的数据。因为两个主服务器需要存储严格一致的数据，所以两台服务器都需要更多的内存以处理持续增长的索引和将足够多的数据存储

在缓存中。由于数据集在持续增长，两台主服务器也同样需要持续增强（通过垂直伸缩的方式）。

除了主-主复制，还可以构建 MySQL 环形复制。这时，主服务器不止两台，而是由三台甚至更多台服务器构成一个环状结构。这玩意看起来不错，实际用起来却很糟糕。图 5-7 所示为这种拓扑结构。

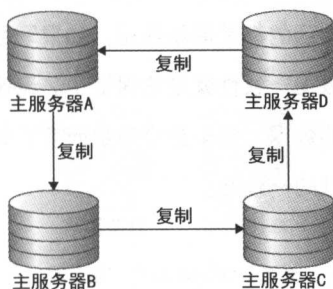


图 5-7 MySQL 循环复制

环形复制不但无助于系统伸缩，由于所有的主服务器都需要执行相同的写操作，还会导致系统的可用性降低，失效恢复更困难。主服务器越多，其中某台服务器宕机的概率也就越高，同时，环状拓扑结构也使得替换服务器更加困难。^[136]

复制延迟用来度量究竟某台从服务器落后其主服务器多少。任何时候，在一台主服务器上执行一个写操作，只要事务提交了，这个变化就立刻在这台机器上可见。虽然数据在主服务器上已经更新了，并且也可以正确读取，但是直到更新复制到从服务器并且被执行后，才能在从服务器上看到这个变化。即使是系统部署的网络环境比较好，这个复制延迟可能也需要一秒。意思是，任何写入主服务器的数据，需要在一秒后才能在从服务器读到。

另一个有意思的事实是环形复制会额外增加复制延迟，每次写操作都要从一台主服务器传到另一台主服务器，直到整个环都完成复制。例如，假设每台服务器的复制延迟是 500ms，有 4 个节点环形复制就需要 1.5s，每次更新复制到其他节点的时间比以前多 3 倍。

提示

需要指出的是主-主结构或者环状结构会使系统更难于理解, 因为从语义上讲, 系统没有一个单一的源。对于一般的主从复制, 对主服务器的查询得到的总是最新的数据。主从方式下, 只有一个主服务器, 不会读到旧数据。写操作会在服务器之间同步, 但是应用只会写到同一台服务器上。这样就可以确保任何时间从主服务器上读数据, 读到的都是最新数据。如果写数据到多台服务器, 服务器之间再异步传输数据, 数据一致性就没法保证了。无论访问哪台服务器, 都有可能读到一些未及时更新的数据。对于整个系统而言, 如何保证一致性, 本章会讨论这种被称作最终一致性的一致性。

复制的挑战

有一件事情需要注意, 就是使用复制进行伸缩的时候, 只是对读操作进行了伸缩, 也就是说, MySQL 数据库的写操作能力并未改善。无论使用哪种拓扑结构, 都没有办法伸缩写操作, 所有写操作全部都落在了一台机器上 (如果有多台主服务器, 就会落在每一台机器上)。视具体业务场景, 复制也许对高可用或者其他目标而言有效, 但是对于一个写操作压力很大的应用, 复制对于分摊写负载压力没有什么帮助。

从另一个方面看, 复制对于读操作压力很大的应用会有很大帮助。如果应用的读操作远多于写操作, 复制就是一种很好的伸缩性手段。这样, 不止一台服务器处理查询请求, 而是多台存储同样数据的服务器一起分摊查询访问压力。通过添加更多的从服务器可以实现更强的读负载承受能力。一台主服务器可以连接的从服务器是有限制的, 不过可以通过配置多级复制添加更多从服务器。随着复制的级数越多, 复制延迟也越大, 数据变更需要扩散到的服务器也越多, 当然读操作的处理能力也越大, 具体如何权衡需要仔细斟酌。图 5-8 所示为多级复制的部署方法。

还有, 复制是一种伸缩只读客户端并发度及只读 QPS (每秒请求数) 的方案, 但不是一种对数据存储规模进行伸缩的方案。例如, 如果你想伸缩数据库以支持 5000 并发只读连接, 那么添加更多的从服务器或者使用更多的缓存就能实现。然而, 如果你想伸缩存储能力达到 5TB 的容量, 复制就无能为力了。显然复制就是把同一份数据复制到不同机器, 主服务器和所有从服务器都存储着相同的数据, 所以有再多的机器, 整体的存储能

力也不会提升。换句话说，每一台机器都要读、写、索引、查找整个数据库的所有数据。

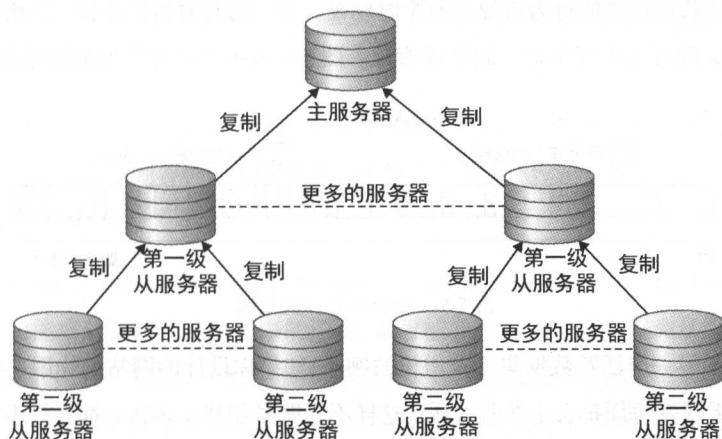


图 5-8 多级 MySQL 复制

活跃数据集是指被应用频繁访问的那些数据。活跃数据集通常很难准确度量，因为数据库不会直接报告自己的哪些数据是活跃的。一个考虑活跃数据集的方法就是假想，数据库里哪些数据需要在某个时间窗内从磁盘读取或者写入磁盘，这个时间可以是一个小时、一天或者一周。

考虑数据访问模式和活跃数据集的大小是一件非常重要的事，因为处理活跃数据是伸缩性的一个常见难题。有大量的不活跃数据可能会增加数据库索引的大小，但是如果不需要不停地访问这些数据，其实对数据库的压力不是很大。活跃数据则相反，需要频繁被访问，数据库需要在内存中缓冲这些数据，或者需要从磁盘中读取这些数据，这通常都是数据库的瓶颈。当活跃数据集本身比较小的时候，数据库可以把大部分（或者全部）活跃数据都缓冲在内存中。随着活跃数据不断增加，数据库需要加载更多的磁盘数据块，因为内存中已经存不下这么多的活跃数据了。到了某个点，缓冲会变得没用，所有的数据都要通过磁盘 I/O 进行随机访问来读取。

为了更好地解释活跃数据集的原理，我们举个例子。假设你有一个电子商务网站，你可能会通过数据表存储交易信息。这类数据通常在购买行为发生后不久被频繁访问，但随着时间的推移访问越来越少。过了几天或者几周，也许还会访问这些交易信息，更新物流

信息或者退款,不过再往后,除了做一些报表统计之类的事情,这些数据基本就不会再访问了。这类活跃数据集的行为就像是有个时间窗一样。随着时间的推移,活跃数据集会增长,但是不会随着每天订单量一起累计增长。图 5-9 所示为这种类型的数据访问情况。

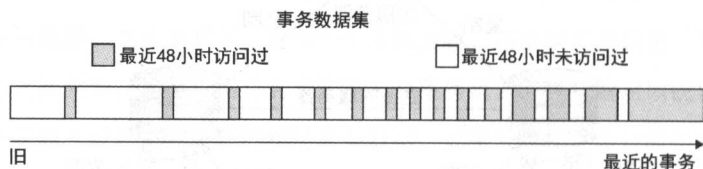


图 5-9 活跃与非活跃数据

我们再看另一个活跃数据集无限增长的例子。如果你设计的网站是让用户收听在线音乐,用户每天或者每周都会上线听音乐。这样不管账号创建了多少,都会被不停登录,请求访问播放列表。随着用户的增长,活跃数据集也不断增长,没有办法删除任何过期数据,系统必须维护这些活跃数据集。本章将讨论如何处理活跃数据集,我们现在需要记住的是,复制不是处理这个问题的合适方法。

另一件需要注意的事情是,当 MySQL 部署为主从复制结构的时候,从服务器返回的数据可能是过期的数据。MySQL 复制是异步的,主服务器上发生的更新需要过一段时间才能复制到从服务器上,所以如果你在主服务器上写数据,同时又去从服务器上读数据,那么很容易就读到脏数据。实践中,根据具体情况不同,复制的延迟,读写请求的时间差,不同服务器的处理时差,应用可能会读到最新的数据,也可能读到脏数据。

正常操作情况下,复制延迟不会超过 500ms,不过这并不意味着,数据同步总能在半秒内完成。MySQL 的复制操作由从服务器上的一个线程单独完成,复制延迟可能会突然增加。主服务器上,所有的更新都是并发执行的,从服务器上,更新则是串行执行的,同时只做一个更新。这是 MySQL 复制最常见的故障原因,如果执行一个耗时较长的更新操作,比如更新表结构操作,会阻塞其余所有表的更新操作,直到更新表结构完成,可能需要花费几秒钟、几分钟,甚至几小时。

如何避免这些因为复制时间导致的问题?一个办法是将写操作的数据缓存在客户端一侧,这样应用就无须到数据库上读取刚刚写完的数据。另一个办法是,将那些对数据敏感的读请求直接发送到主服务器上,这样的数据就一定是最新的数据。还有一种办法,就是尽量减少复制延迟,降低从服务器读到脏数据的概率。举个例子,为了避免更新表结构

引起的阻塞，将主服务器上这一类操作的 binlog 关闭，手工在从服务器上执行。通过这种方式，更新一个大表的表结构将不会导致其他表更新的复制阻塞，而所有服务器最终也会拥有相同的表结构。

不理解 MySQL 复制的复杂性和各种代价可能是致命的，对于一个缺乏经验的数据库管理员而言，可能会遇到各种严重的挑战，必须深入理解 MySQL 和 MySQL 复制，才能安全可靠地用好 MySQL 复制。

中断 MySQL 复制或者其他原因导致数据不一致的情况有好多种。比方说，使用了生成随机数的功能或者执行了某个受限的语法，都可能导致写入主服务器和从服务器的数据不一样，破坏数据的一致性。一旦主服务器和从服务器的数据同步出了差错，你的麻烦就大了，后面所有的更新操作在不同服务器上的执行结果可能都会不同，可以想象，相同的操作得到的结果也不一样。出了这种问题，Debug 也是异常困难，这种 BUG 被称作幽灵 BUG，一个操作在主服务器上明明是正常的，到了从服务器上就抛出错误了，只好停止复制。

虽然有一些开源工具，比如 pt-table-checksum 或者 pt-table-sync，可以协助解决这类问题，但是目前还没有一个自动化的高可用的工具能够简单易行地保障 MySQL 复制。如果复制中断了，就只能自己去搞定，这需要大量的知识、经验和时间。

考虑到管理 MySQL 复制的巨大挑战，所以使用云主机提供的 MySQL 解决方案，比如亚马逊 RDS（关系数据库服务）或者 Rackspace 云数据库，也是一个不错的选项，可以减少很大的负担。特别是对于一家初创期的创业公司，你的主要职责是尽快应付市场的挑战，所以最好是使用云服务，而不是花时间学习自己搞定所有技术挑战。云数据库会提供许多有用的功能特性，比如设置复制、自动备份，以及从服务器一键重启。有些云提供商还会支持更多的高级特性，比如自动失效转移到另一个可用区。不过，复制还是有可能引起同步不一致，所以学习一点 MySQL 的运维知识是有必要的。

虽然本节主要关注的是 MySQL 数据库复制，但是许多知识原理也同样适用于其他各种数据存储系统。复制是数据存储的一种常规实现，异步完成一主多从的数据状态同步。各种存储的实现细节会有不同，挑战有难有易，但是它们都能实现将查询请求负载分布到多台服务器上的伸缩性目标，将慢查询和备份隔离到不同的服务器上。但是，不论是否使用 MySQL、Redis、MongoDB，以及 Postgres 的复制功能，都没有办法实现写操作和数据

存储量的伸缩性。现在我们看一下第二个伸缩性技术——数据分区，也被称作分片。

数据分区（分片）

数据分区是第 2 章提到的三个主要伸缩性技术之一（其余两个是功能分割和增加备份）。数据分区的核心目标是将数据集切分成较小的分片，以便于将它们分散存储在多台服务器上，不让一台机器处理整个数据集。通过将数据集切分成较小的分片，并将这些分片分配给不同的服务器，服务器之间互相独立，不共享任何信息（至少在简单的分片场景下）。没有数据重叠，每台服务器都可独立进行数据操作的权限管理，无须进行服务器间通信，即使有部分服务器故障，也不会影响整个系统的可用性。

人们常常把数据分区称作分片，这个术语的原始出处不太明确，很多人认为最初来源于 20 世纪 90 年代的 Ultima 在线。Ultima 在线是第一个大规模多用户在线角色扮演游戏，其数据规模庞大，开发者决定将游戏里的世界分割到不同服务器里（当时就称作分片）。在这个游戏的世界里，开发者将一种和现实世界不一样的碎片化世界观组织在游戏的故事线里，用以解释这些独立的平行世界。每个世界都是独立的，角色被绑定到特定的某个分片上，不能跨分片交互。

无论其原始出处如何，分片都可以用运输玻璃片来比喻。考虑到尺寸和重量，玻璃片越大，运输和处理玻璃片的难度越大。一旦将这些玻璃片切分成小片，运输起来就容易多了。无论最开始时玻璃片有多大，都可以通过切分将这块玻璃封装在很多个桶，并装在很多个货车上，而不用一次就对付那么大一整块。如果将玻璃片变成数据集，桶变成运行数据存储系统的服务器，分片就是将一整个单一的大数据集切分成很多小的部分，然后放在很多个服务器上。

选择分片键

分片的核心原理就是将数据以某种方式切分，以便每台服务器都只存储一部分数据。读写数据时，只需要和存储着需要的数据的服务器通信，而不是向所有服务器发送请求。分片键的主要用途就是定位分片，直接访问特定的服务器，而不是和所有的服务器通信。

分片键用来决定要访问的数据该由哪台服务器负责。分片键的用法有点类似于对象缓存的键。要从缓存中读取数据，就必须知道缓存的键值，这是定位

数据的唯一方法。分片键也类似，要访问数据，就需要用分片键找到数据在哪台服务器上。一旦知道了数据在哪台服务器上，就可以和这台服务器通信，发送查询请求。

为了更好地说明分片的原理，我们再举一个电子商务网站的例子。假设你开发了一个大规模电子商务网站，并把所有的用户数据都放到了一个 MySQL 数据库里，这个数据库部署在一台服务器上，如图 5-10 所示。在这种情况下，访问用户数据的时候，不需要选择服务器，因为所有的数据都只在一台服务器上，一台机器上存储了所有的数据。

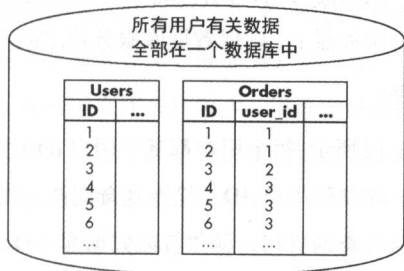


图 5-10 未分片的用户数据库

如果你想对数据规模进行伸缩，使用多台服务器进行存储，可以选择分片技术将数据分布在多台 MySQL 数据库服务器上。使用分片技术的时候，需要用某种方式将数据切分到若干独立的分区上。例如，对于传统在线商店，用户之间彼此不需要交流，因此可以按用户进行分区，而不会牺牲功能。通过这种方式，可以很容易地将用户分布在若干台服务器上，当需要对某个用户进行读写操作的时候，只需要连接特定的服务器就可以了。

选择了数据切分的方式，就需要选择分片键。如果分片基于用户，可以使用某个能够唯一标示用户的字段作为分片键，比如账户 ID（也就是用户 ID）。选择了分片键以后，还需要选择一个算法，将分片键映射到某个实际的服务器编号上。简单起见，我们讨论只有两个分片的情况。在这种情况下，可以将所有偶数用户都分到分片 1，所有奇数用户都分到分片 2。图 5-11 所示为用户数据与服务器编号之间的映射过程。

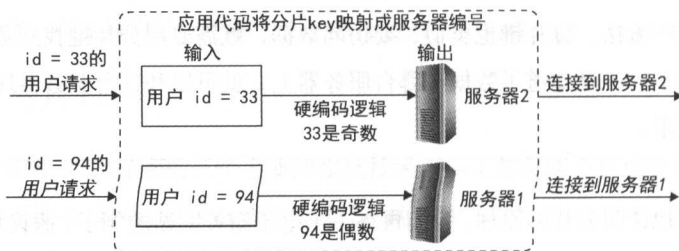


图 5-11 分区 key 映射到服务器编号

通过选择分片键和映射算法，将数据分布到多台服务器上。每台服务器上存储差不多数量的数据，当有新用户到来的时候，同样根据用户 ID 为其分配一台服务器。此外，每一个数据都只存在于一台服务器上，确保数据库服务器之间不共享信息，每台服务器只对自己存储的数据拥有权限。

分片后的数据库如图 5-12 所示。每个用户都基于用户 ID 分配到某台数据库服务器上。需要访问用户数据的时候，先拿到用户 ID，检查其奇偶性，找到对应的服务器。得到服务器编号后，连接服务器执行查询语句，就跟普通数据库一样。事实上，MySQL 不需要任何特别的配置，MySQL 不需要知道自己是一个普通数据库存储整张数据表，还是一台分片后的数据库存储了部分数据。所有的分片逻辑都只存在于应用程序之中，所有分片数据库的表结构都是完全相同的。

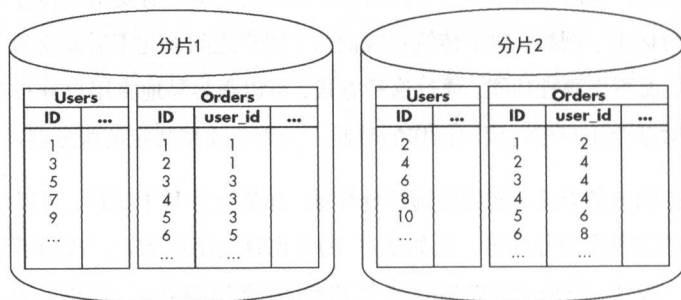


图 5-12 使用分片的用户数据库

仔细看图 5-12，可以注意到订单 ID 在各个分片之间并不唯一。订单 ID 是自增长的，各个分片数据库之间彼此并不知道，在每个数据库上都会有相同的 ID。在有些情况下，这是可以接受的。如果你需要全局唯一订单 ID，可以使用自增长偏移方法使得每个分片都生成不同的主键。

提示

分片可以在数据层的顶端，应用层实现。需要做的就是找到一种方法对数据进行切分，使数据分布在不同服务器上，再找到一种方法对查询进行路由，找到正确的服务器。数据存储本身并不需要支持分片，不过有些数据存储本身支持自动分片，数据分片对使用者透明。本章会讨论自动分片技术。

在这个例子中，我们使用用户 ID 作为分片键，通常情况下，人们会创建更多的小分区，而不是少量的大分区。将数据分片到少量的大分区没办法使用更多的服务器。例如，根据用户国家进行分片，使用国家代码作为分片键，将国家代码映射到服务器编号。这种方法看起来不错，事实上却不是那么回事。根据国家进行分区，可能会导致数据分布不均衡，有的国家用户比较多，有的国家用户比较少，很难保证数据分布及负载的均衡性。把数据切分到若干大分区，最终有可能使某个分区变得特别大，单一一台服务器根本就放不下。例如，来自美国的用户数量快速增长，超过了单一服务器的极限，分片的目的根本就没有达到。图 5-13 所示为根据国家代码进行分区导致的服务器超载的情况。国家是平等的，但是国家人口却不相等。

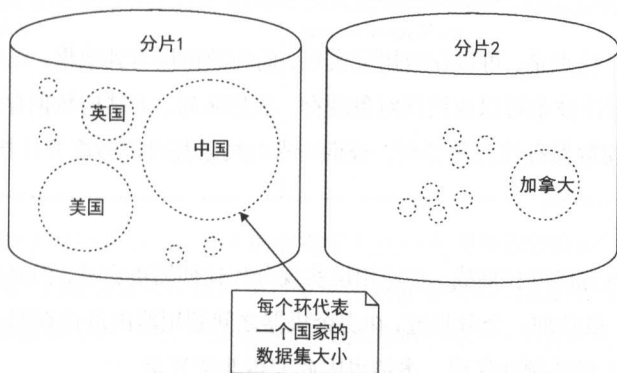


图 5-13 不均匀的数据分布

分片的时候尽量将数据切分成比较小的数量放在更多的分区中，有助于将数据均衡地分布在多个服务器上。一般情况下，并不能确保所有的分区数据量都是一样的，但是如果分区的数量足够多，每个分区存储的数据量又不太多时，整个集群仍然能够表现出一个不错的数据分布均衡性。

分片的优点

如你所见,如果使用得当,数据分片最大的优点就是可以根据数据量对数据库进行水平伸缩。

对于一个真正的水平伸缩系统,所有的组件都需要能够做到水平伸缩。如果不进行分片,无论怎么做,都会触及 MySQL 处理能力的极限。或迟或早,数据量大小会超过单一数据库能够存储的极限,或者能够处理的并发连接的极限。随着读写数据量的增加,磁盘 I/O 吞吐也会遇到极限(具体要看一台服务器能够挂载的硬盘数量有多少)。

通过使用应用级的分片,不需要一台服务器处理所有的数据。你可以部署多台 MySQL 服务器,每台服务器都配置普通容量的内存、磁盘和 CPU,每台服务器都只负责存储、查询、读写整个数据集的一部分数据。通过增加服务器可以对整个集群的处理能力进行进一步伸缩,而不需要增强每一台服务器的配置。

由于分片是将数据切分成互不关联的若干子集,所以可以实现无共享架构。服务器之间无须进行通信,不需要进行集群级别的同步,也就不会出现相应的同步阻塞。每台服务器都是独立的,跟普通 MySQL 服务器没什么两样,管理、维护、优化、伸缩也和普通 MySQL 实例一样。

分片的另一个优点是,可以在应用层实现,而在数据层看到效果,无论数据存储本身是否支持分片。分片技术可以应用到对象缓存、消息队列、无结构数据存储,甚至文件系统。任何对大规模数据有持久化管理、查询请求的场景都可以考虑分片技术。

分片的挑战

当然,分片也有代价和挑战。在应用层实现分片让伸缩更容易,但是也相应地增加了复杂度和工作量。虽然加一个分片键,在多个机器之间利用路由进行查询,听起来很简单,但实践中,需要大量的额外代码,事情也因此变得非常复杂。

应用层分片的一个主要制约是无法执行多分片的联合查询。任何时候想执行一个查询,都只能在每一个分片上分别查询,然后把各个分片的返回结果在应用层合并起来。有些情况下,这种合并比较容易,有些情况则非常困难。

为了更好地说明情况,我们参考一个例子。假设你有一个电子商务网站,基于用户 ID 进行数据分片,将数据切分在多台服务器上,那么你可以很容易地访问特定用户的数

据，但是没办法执行涉及多个用户的查询。如果你想查找过去 7 天的最热卖商品，就需要先在每个分片上执行查询，然后在应用里计算出正确的结果。即使是这么简单一个场景，也还是很容易判断失误，写出错误的代码，主要还是我们并不习惯使用分片数据。如果所有的数据都放在一台机器上，那么获得最热卖商品的查询语句如代码清单 5-1 所示。

代码清单 5-1 一个简单的 GET 请求示例

```
SELECT item_id, SUM(amount) total
FROM orders WHERE order_date > '2014-11-01'
ORDER BY total LIMIT limit 1;
```

想象一下，也许你觉得只要要在每台服务器上执行相同的查询语句，然后挑出最大值就可以了。非常不幸，这样做是错误的。假设你有两台服务器，每台服务器的销售数据如表 5-1 所示，那么这个代码返回结果显然不正确。在每个服务器上执行代码清单 5-1，然后挑出一个最大值，得到的结果是分区 B 上的 item_id=2，值是 16。再多看看这张表，就会发现 item_id=5 的值更大，值是 23。

表 5-1 每个分区的汇总数据

分区 A 的顶级销售		分区 B 的顶级销售	
item_id	Total Sales	item_id	Total Sales
4	13	2	16
5	12	3	14
1	10	5	11
...		...	

从上面的例子可以看出，在分片的数据集上执行查询操作很麻烦。虽然代码清单 5-1 的例子非常简单，但还是需要应用对每台数据库服务器获取大量的数据，才能计算出正确的结果。如果查询再复杂一点，难度会呈指数级增加，执行一个复杂的数据报告类查询是极其有挑战的。

术语 ACID 事务指的是大多数关系数据库引擎都支持的一种事务属性。A 表示原子性(Atomicity), C 表示一致性(Consistency), I 表示隔离性(Isolation), D 表示持久性 (Durability)。原子性事务要求整体执行，要么全部完成，要么全部拒绝恢复原样。一致性保证只要事务完成，一定从某个一致性状态转换到

另一个一致性状态。隔离性保证事务可以并行运行而不会互相影响。最后，持久性保证在返回到客户端之前，数据可以持久存储，事务一旦完成，即使是服务器宕机，数据也不会丢失。当人们说某个数据存储支持 ACID 事务的时候，就是说在数据存储上执行的每个事务都保证满足 ACID。

跨多台服务器进行分布式数据访问的一个副作用是没办法使用数据库的 ACID 事务。虽然仍然可以使用每个分片上的 ACID 事务，但是跨多台机器的 ACID 就没办法保证了。想在这种情况下实现 ACID 事务需要使用分布式事务，非常复杂，执行成本也极高（很多开源数据库引擎，比如 MySQL，不支持分布式事务）。例如，如果要更新某个用户的全部订单数据，那么就在某台服务器上完成全部的操作，这时可以使用数据库的 ACID 事务。但是，如果想要更新某个商品的全部订单，就需要将命令发送给全部的服务器。这种情况下，就没办法保证所有的数据操作全部成功，或者全部失败。可能在分片 A 上执行成功了，但是在分片 B 上执行失败了，这时候又没办法回滚分片 A 上的操作，因为事务已经完成了。

利用应用层对数据分片需要面对的另一个挑战是：随着数据的增长，如何增加更多的服务器。不同的分片键与服务器编号映射方法，添加服务器到分片集群的难度也不同，有的方法难度极大。

在本节的开始，介绍了分片键用于映射服务器编号。最简单的映射方法是取模。本节的第一个例子中，将奇数用户 ID 的数据分到分片 A，偶数用户 ID 的数据分到分片 B，这实际上就是用 2 取模。

取模 $\text{modulo}(n, x)$ 就是 n 除 x 的余数。可以将任何整数映射到 0 到 $n-1$ 的范围内。比如，你有 6 台服务器，那么你可以用 $\text{modulo}(6, \text{userId})$ 根据用户 ID 计算出服务器编号。

这种基于取模算法的映射方法有个问题，每个用户被映射到某个特定的服务器上是基于服务器的数量。当服务器的数量改变时，大多数的用户服务器映射关系也改变了。例如，如果你有 3 台服务器，编号是 0, 1, 2，那么用户 ID=8 的时候，将会被映射到最后一台服务器 2 上， $\text{modulo}(3, 8)=2$ 。如果新加一台服务器，服务器数量变成 4，服务器编号为 0,

1, 2, 3, 对用户 ID=8 执行相同的取模算法, 将会得到完全不同的结果, $\text{modulo}(4,8)=0$ 。

如你所见, 增加服务器是一个极大的挑战, 需要在服务器之间迁移大量的数据。需要持续计算哪个数据迁移到哪台服务器。当对整个系统做水平伸缩的时候, 伸缩过程应该是低成本且简单的, 所以我们需要找其他的方法。

避免每次添加服务器都要迁移数据及再分片的一个办法是将所有的映射关系保存在一个独立的数据库里。这样不需要基于某个算法计算映射关系, 可以用分片键查找服务器编号。在电子商务的例子中, 我们用一个独立数据库存储用户和服务器编号的映射关系。如图 5-14 所示, 应用查找存储在数据库中的映射关系, 为了加速查找, 应用会缓存部分映射关系。

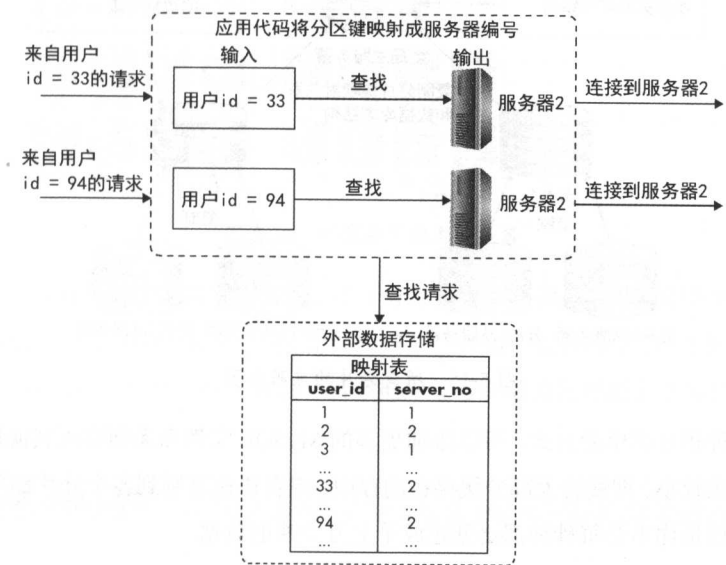


图 5-14 外部映射数据存储

将映射数据保存在数据库的好处是在不同分片之间迁移数据变得更容易。不需要一次迁移所有数据, 可以进行增量迁移, 一次迁移一个账号。要迁移一个用户, 先要锁定这个账号, 然后迁移数据, 再解锁账号。可以在晚上进行迁移以降低对系统的影响, 也可以同时迁移多个账号, 只要没有数据重叠就行。

将映射关系存储在数据库中,还可以增加弹性,可以挑选用户迁移到选择好的分片上。根据应用需求,可以将最大或者最忙的用户迁移到特定的服务器上,给它们更好的硬件配置。同样,如果你觉得用户活动太频繁不是一件好事,可以将那些消耗了太多计算资源的用户放在一起作为惩罚。

映射关系也需要一个地方存储,可以使用 MySQL,也可以用其他存储系统。如果选择 MySQL 存储,可以部署一个 MySQL 主服务器存储映射表,然后复制数据到各个分片。这种部署模式下,任何时候如果要创建一个新用户,都要写入数据到主服务器。然后将用户条目复制到各个分片,在任何分片上都可以进行映射关系查询,如图 5-15 所示。

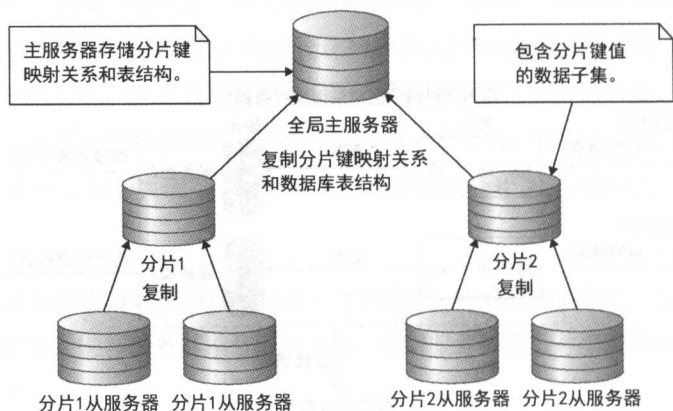


图 5-15 所有分片的主服务器

这是一种相对简单的方式,可以添加更多的 MySQL 实例而无须引入其他技术。由于映射数据集比较小,即使将大部分缓存在内存中,或者快速复制到各个分片都不是什么太难的,前提是你不会每秒钟都要创建成千上万个映射数据。

根据系统的基础设施情况,可以选择增加一个 MySQL 实例存储映射数据,如果已经使用了其他的可伸缩数据存储(将在本章后面部分讨论),也可以考虑将映射数据存储在这类可伸缩数据存储中,无须将数据复制到各个分片。将映射数据存储另一个数据存储中会增加整个系统的复杂度,虽然还得部署、管理、伸缩另外一个数据存储实例,但如果系统已经在使用这样的数据存储,那么相对就容易多了。

幸运的是,还有一种分片方案可以降低再分片的麻烦,成本相对也较低,复杂性也不大。这种方案还是用取模方法将分片键映射到数据库编号,但是这里的数据库只是一个逻辑

辑上的 MySQL 数据库，而不是一个物理机器。首先，先确定开始时的机器有多少个，然后再预测将来需要的机器有多少个。

例如，你估计开始的时候需要两台机器，而将来最多也不会超过 32 台机器（32 个分片）。这样的话，可以在每台物理服务器上创建 16 个数据库。将服务器 A 上的数据库命名为 db-00...db-15，将服务器 B 上的数据库命名为 db-15...db-31。然后在所有数据库上都创建相同的表结构，部署架构如图 5-16 所示。

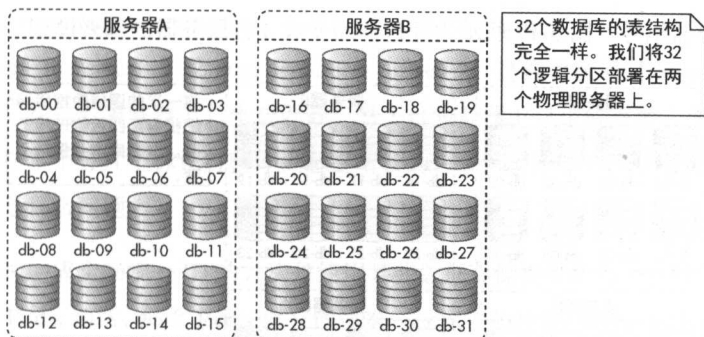


图 5-16 所有分片的主服务器

同时，还要在代码中实现映射方法，基于分片键得到数据库编号和服务器编号。实现一个 `getDbNumber` 方法，将分片键（比如用户 ID）映射为数据库编号（本例中是 0-31），再实现一个 `getServerNumber` 方法，将数据库编号映射为物理服务器编号（本例中为 0 或 1），如代码清单 5-2 所示。

代码清单 5-2 映射函数

```
/**
 * Returns a logical database number based on the value of
 * the sharding key.
 * @param int $shardingKey
 * @return int database number
 */
function getDbNumber($shardingKey) {
    return $shardingKey % 32;
}

/**
 * Returns a physical server number based on the db number.
 * @param int $dbNumber
```

```

* @return int physical server number
*/
function getServerNumber($dbNumber){
    return $dbNumber < 16 ? 0 : 1;
}

```

这样就可以部署应用开始运行了。当数据库增长需要进行伸缩的时候,只需要简单地将物理服务器增加一倍,将原来每台服务器上的一半的逻辑数据库迁移到新的服务器上。同时修改 `getServerNumber` 方法的映射关系,根据新的逻辑数据库与物理服务器关系返回正确的服务器编号。图 5-17 所示为经过伸缩变成 4 台服务器的部署模型。

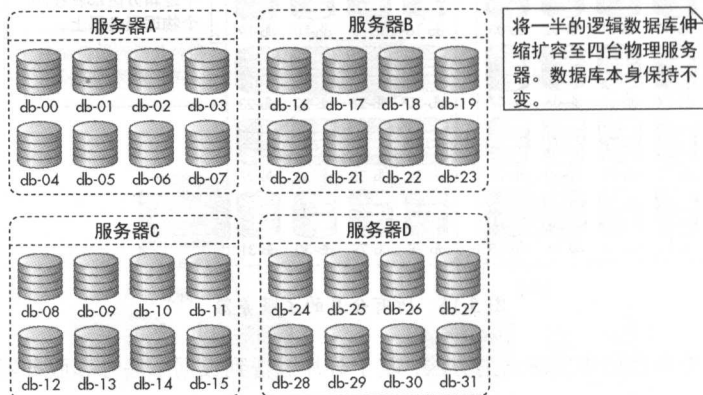


图 5-17 伸缩扩容后的多数据库分区方案

虽然每个服务器安装多个数据库相对前面那种最简单的分片方案多了一点点复杂性,但是当系统需要伸缩的时候获得了更加强大的弹性。只需要简单复制数据库,修改几行代码,就可以使整个系统的存储能力翻倍。操作起来更容易更安全,整个迁移过程无须更改、插入、删除数据,需要做的仅仅是将整个 MySQL 数据库从一个服务器移动到另一个服务器。

这种方案还有一个好处,就是可以在非常短的停机时间内快速完成伸缩。只要规划得当,可以在几分钟的停机时间内完成一次伸缩活动。实践中,伸缩过程可能如下:

- 首先,将新加入的服务器设置为当前服务器的从服务器,进行复制。
- 然后,停止写操作一小会儿,等待所有的数据都同步完成。
- 当新服务器的数据赶上主服务器、主从数据一致的时候,关闭新服务器的复制。

- 修改应用的配置，使用新服务器进行读写。

应用级分片面临的另一个挑战是很难针对所有的分片生成唯一标识符。有些数据存储本身可以生成全局唯一 ID，但是 MySQL 并不支持分片，所以应用需要自己处理这种情况。

如果你并不关心唯一标识符长什么样，那么可以用 MySQL 自增长字段加偏移量的方法，这样每个分片生成的标识符就会不同。对于一个只有两个分片的系统，在一台服务器上设置 `auto_increment_increment=2` 及 `auto_increment_offset=1`，在另一台服务器上设置 `auto_increment_increment=2` 及 `auto_increment_offset=2`。这样每次通过 `auto_increment` 生成值的时候，在一台服务器上只生成偶数，另一台服务器上只生成奇数。使用这种技巧，并不能保证在多个分片上标识符整体上总是增长的，每个服务器都有不同的编号，不过总的说来并不是什么问题。

另外一种生成全局唯一 ID 的方法是使用某些数据存储提供的原子计数器功能。例如，如果你用了 Redis 的话，那么 Redis 就有创建唯一标识符的功能，Redis 的 `INCR` 命令可以对某个计数器的值进行递增并返回。这样的话，多个客户端可以并发请求获得新的标识符，每个客户端得到的值都是不同的，保证数据连续递增且全局唯一。

提示

应用级分片解决这些复杂性的一种方式是使用云主机。例如，Azure SQL 数据库弹性伸缩方案提供一组代码库和服务工具解决分片问题，分片管理、数据迁移、数据映射，甚至跨分片查询等全部搞定。无须自己开发相关代码和支持工具，开发者可以更多关注业务实现。虽然 Azure SQL 数据库是 SQL Server 的一个定制版本（而非 MySQL），但确实是云主机提供可伸缩 SQL 方案的一个极好例子。

如你所见，分片的主要挑战来自于应用层。我们快速过一下整体架构，看一看如何使用 MySQL 打造一个集成了复制、分片、功能分割等各种伸缩性特性的系统。

集成架构

前面我们讲过，伸缩性技术可以浓缩成如下三种模式：增加备份、功能分割，以及数据分区。所有三种模式都适用于基于 MySQL 构建的系统的伸缩性。再次想象下那个电子

商务网站的例子。这次，我们考察的范围更大一些，我们探讨各种不同伸缩性技术具体如何实现。

如果构建一个电子商务系统，简单一点的方法，就是把应用的全部功能都部署在一起。稍微复杂一点，将 Web 服务分离出来，和一台 MySQL 通信，实现数据的持久化需求，如图 5-18 所示。

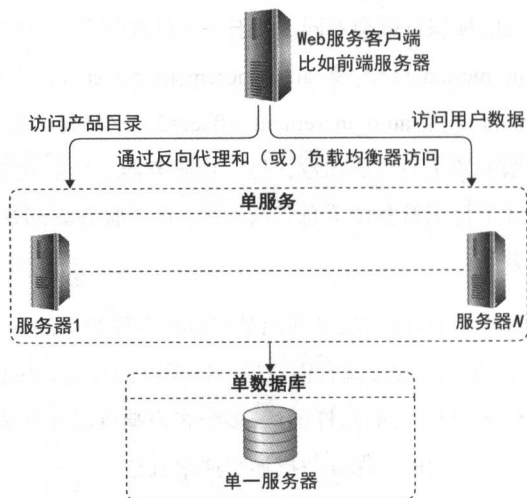


图 5-18 单一服务与单一数据库

假设 Web 服务是无状态的，可以实现 Web 服务器的伸缩，但是没办法增加数据库服务器。如果应用的读操作远远超过写操作，那么可以通过增加只读的复制服务器实现伸缩。所有的服务器和主服务器存储有完全一样的数据，这就是通过增加备份实现伸缩。这种配置的系统如图 5-19 所示。

现在，如果系统需要伸缩，那么可以通过功能分割的方式切分出两个功能组件。比如可以将以用户为中心的数据放在一个数据库上，其余的数据放在另一个数据库上。同时，Web 服务的功能也拆分成两个独立的 Web 服务 ProductCatalogService 和 CustomerService。ProductCatalogService 负责管理和访问产品、类目、促销相关的信息，CustomerService 负责用户账号、订单、发票、购买记录相关的信息。功能分割后，系统架构如图 5-20 所示。

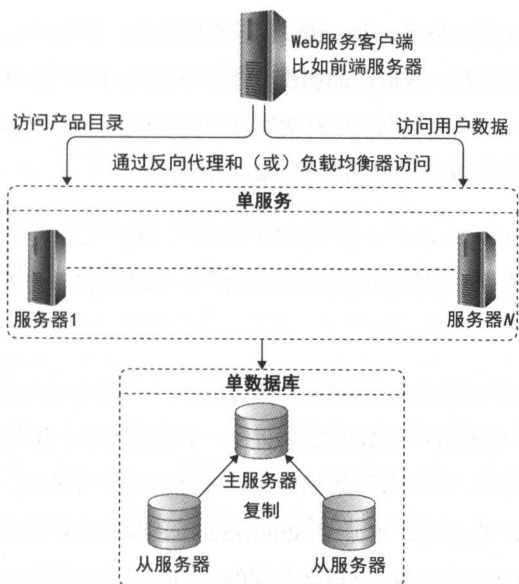


图 5-19 通过增加备份伸缩类目

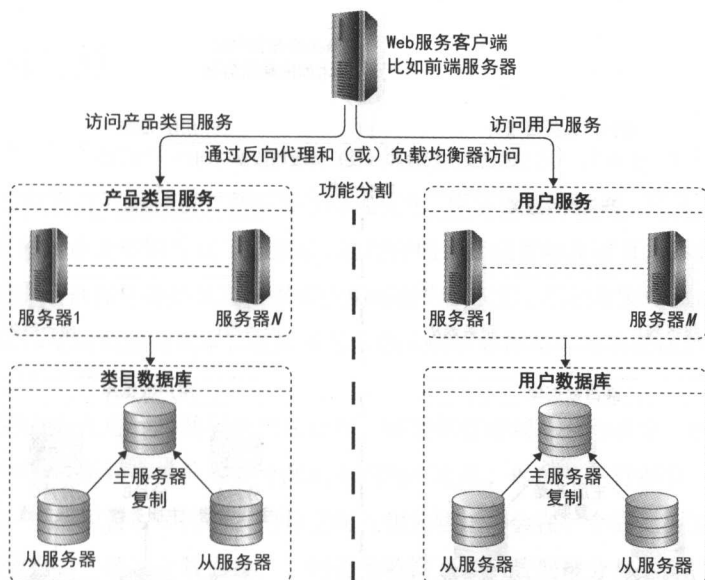


图 5-20 两个 Web 服务及两个数据库

将 Web 服务拆分成两个低耦合的服务，可以更方便地对其进行独立伸缩。可以使用

MySQL 复制对 ProductCatalogService 进行读操作的伸缩, 因为产品类目信息主要是只读操作。用户会搜索产品数据, 获取产品明细, 查看不同类目的产品列表, 但是几乎不会通过 ProductCatalogService 修改数据。只有卖家才可能修改产品类目信息, 不过相对其他各种产品类目操作, 这些修改操作只是非常少的一部分。

相反, CustomerService 会产生相当多的写操作, 用户大部分的这类操作都需要持久化到数据存储。每个操作, 比如添加商品到购物车, 处理支付, 请求退款, 都需要对数据库进行写操作。

由于产品类目主要是只读的, 数据量也相对较小, 可以选择增加只读复制进行伸缩(增加备份的伸缩方案), 将所有产品数据都存储在一个数据库中, 添加复制服务器提供更强大的只读处理能力。相反, 用户数据集相对比较大又主要是写操作, 就需要利用分片的方案进行伸缩(数据分区的伸缩方案)。CustomerService 不需要复制进行伸缩, 但是每个分片最好保留一个只读的从服务器, 以提高存储的可用性及实现数据备份的目的。系统架构如图 5-21 所示。

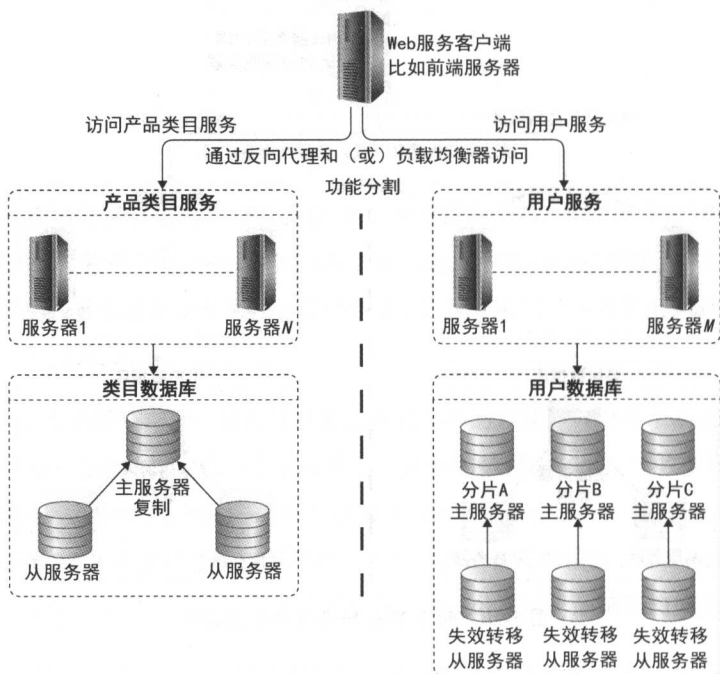


图 5-21 使用所有三种技术

在这个案例中,复制使用 MySQL 的复制功能实现,分片通过应用层实现。应用要在 cookie 或者 session 中存储用户 ID,这样每次发送给 CustomerService 的请求就会包含用户 ID (这个 ID 可用作分片键)。基于分片键,Web 服务能够连接到正确的分片服务器(也就是一个普通的 MySQL 实例),进而执行相应的查询操作。如你所见,你可以混合使用两种伸缩性技术来满足使用 MySQL 数据库系统的增长和发展。

虽然应用层分片是增加系统 I/O 处理能力及存储更多数据的一种好办法,但是这种方案会带来很多挑战。代码会变得更加复杂,跨分片的查询是所有工程师的痛,还需要仔细选择分片键,增加硬件时的数据迁移也是个巨大的挑战。

复制可以改善只读操作的吞吐能力,提高存储的可用性,不过它也有自己的难处,在确保数据一致性和失效恢复方面目前做得并不好。需要应用自己关注复制的技术细节,应用要能够正确处理复制延迟,并将查询发送到正确的服务器上。

幸运的是,MySQL 及其他一些关系数据库并不是存储数据的唯一选择。我们将会了解其他存储引擎,这些引擎能够更好地处理伸缩性和可用性的问题。

NoSQL 伸缩性

一直以来,关系数据库对 Web 应用的伸缩性就是一个痛点。计算机科学专业的学生,几十年来一直被教育数据管理要遵循事务的规范化,保证数据一致性。有些开发者做得更加极端,把大量的业务逻辑写在了数据层,通过存储过程或者触发器直接在数据库服务器上执行业务逻辑。这在传统的关系数据库应用场景中很常见,但如果需要对这样的数据库进行伸缩以适应更高的应用需求,就需要很多强大的服务器及一大堆价格昂贵的许可证。

数据规范化是一种数据结构化过程,将数据整理到不同的表中。规范化过程的一种方式是将数据分解,分配到不同的字段里,保证每一行都有一个主键作为唯一性标识符,不同的表的行之间互相关联,不会在一个表里重复另一个表里的数据,一则减少数据量,二则避免数据冗余。数据被分割的规模越小,就越有利于索引和搜索。数据规范化也会提高数据的集成度,应用需要更新数据的时候,只需要更新一份数据(其他表的行会关联这一行数据,而不会自己复制一份)。

随着互联网业务的持续发展,技术的持续进步,关系数据库一统江山的局面大概在 2000 年后开始被打破。企业需要伸缩性更强大的数据管理系统,急切寻找一种创新性的技术去管理数据。企业不再纠结于完全的 ACID 事务保证及复杂的分布式事务实现,亚马逊、Google、Facebook 决定自己实现一个更简单的数据存储系统。这些数据存储系统不再支持 SQL 语言、复杂的存储过程或者触发器,而是提供真正意义上的完全水平伸缩和高可用特性,这些特性是过去的关系数据库不曾做到的。

这些数据系统已经证明非常成功,一些世界上最大的互联网创业公司陆续发布技术白皮书公开他们的创新成果。Google 发布了 Google 文件系统 (GFS)、^[44] MapReduce,^[w1] 以及 BigTable,^[w28] 随后亚马逊公开了一个极其重要的数据存储系统 Dynamo。^[w39] 到了 2010 年,这些早期的探索者制定的设计原则和方案逐渐被更多的后来者遵循,涌现出一大批类似产品,Cassandra、Redis、MongoDB、Riak、CouchDB 等,NoSQL 时代来临。

NoSQL 是一个很宽泛的术语,很多类型的数据存储系统都给自己贴上这个标签以示自己与传统的关系数据库不同。这些数据存储系统通常不支持 SQL 语言,所以叫 NoSQL。

这些新技术在处理持续增长的数据方面如此成功的一个主要原因是为了支持伸缩性,系统做出了巨大的取舍。

设计一个 NoSQL 数据存储系统最先要做的决策就是,这个系统最重要的特性是什么(例如可用性、响应延迟、一致性、易用性、事务保证或者其他一些伸缩性维度)。一旦决定了特性的优先级,就可以针对重要性做出权衡。同样,如果你要进行开源 NoSQL 产品选型,也要先决定特性的优先级顺序,然后寻找一款合适的产品,而不是找一个产品企图满足所有方面的需求。如果你想在所有 NoSQL 产品里找到一款“更好的 NoSQL”,恐怕是要失望了,所有的 NoSQL 数据存储系统都是在牺牲了其他一些特性后实现对最重要特性的支持的,如果你要构建一个可水平伸缩的数据存储层,你要做好准备接受这些被牺牲掉的特性。

传统上,数据库设计者们并不觉得自己需要做出权衡和牺牲,直到 Eric Brewer 提出著名的 CAP 原理^[w23-w25],指出不可能构建一个分布式系统能够同时满足一致性、可用

性，以及分区耐受性。在这个原理中，一个分布式系统包含若干节点（服务器），通过网络连接实现这些节点之间的通信。一致性是说这些节点在同一时刻看到的数据是一样的。可用性是指当某个节点失效的时候，有其他节点可以为客户端提供同样的服务。分区耐受性是指当网络失效，节点间无法通信的时候，系统仍然可用。

提示

CAP 原理其实很难理解，CAP 原理里面定义的一致性和 ACID 的传统定义并不一样。在 CAP 原理里面，一致性保证同一份数据在同一个时刻在所有节点是同样可见的。意思是，所有的状态变更都是串行发生的，一个发生后另一个再发生，而不是并行同时完成。换句话说，需要一种跨 CPU 和服务器的机制保证返回的数据是最新的数据。ACID 里面，一致性关注的是数据的关系，比如外键约束的一致性，以及唯一性等。

由于所有可用的节点要处理所有到达的请求（可用性），在同一时刻需要响应相同的数据（一致性），所以没有办法在网络中断的情况下保证数据能够正确的复制和传播。图 5-22 所示为一个和 CAP 原理冲突的假想的数据存储系统。从这个例子中可以看到，网络失效隔断了节点 A 和节点 B。还可以看到，节点 C 也失效了，多个客户端使用不同的节点读写同一份数据。

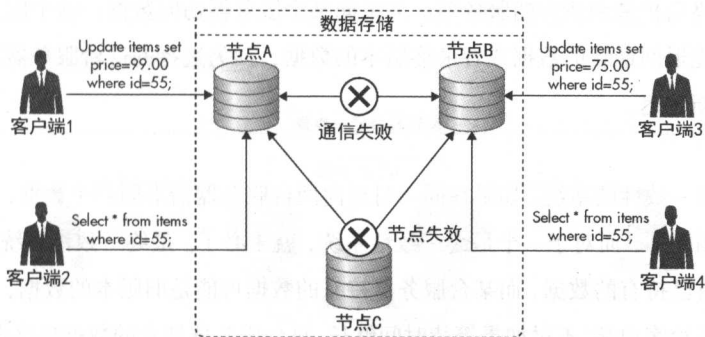


图 5-22 CAP 原理与数据存储冲突

CAP 原理很快就流行起来了，它为新的 NoSQL 数据存储设计者提供了如何做出权衡的判断依据。简单说就是：一致性、可用性和分区耐受性，三个中最多选两个。这句话并

不完全正确,但是为工程师打开了视野,意识到传统关系数据库在保证事务和其他特性的情况下,不可能实现期望的伸缩性。2012 年, Brewer 发表了白皮书 *CAP 12Years Later*, 书中解释了一些关于 CAP 的误解, 以及进行权衡的时候更微妙的一些选择, 而不只是牺牲掉整个一致性或者可用性。不过, 不论准确与否, “三选二” 都成了 NoSQL 的标语, 它释放了一种强烈的信息, 那就是, 可伸缩的数据存储系统需要进行权衡。

最终一致性

根据 CAP 原理的简单版本的建议 (三选二), 构建一个分布式数据存储系统, 对于可用性、一致性和分区耐受性这三个特性要适当放宽。有的 NoSQL 数据存储系统选择牺牲一定的一致性保证, 从而使伸缩性更容易。亚马逊的 Dynamo 就选择了这种策略^[w39]。没有实现强制的完全一致性, 也没有尝试实现分布式事务, 亚马逊认为高可用对于他们的业务是最重要。亚马逊不希望用户在浏览页面的过程中看到一个空白的页面, 也不希望用户的购物车里莫名其妙地丢失了货物。

基于这个优先级设定, 亚马逊为了支持高可用及伸缩性需求, 在其他方面做出了重要的牺牲。牺牲了复杂查询, 简化了数据模型, 并引入了一个最终一致性的概念, 放弃实现一个全局的一致性系统。

最终一致性是系统的一个属性, 不同节点持有不同版本的数据, 但是数据更新最终会扩散到所有的服务器。如果向某个服务器访问数据, 这个服务器并不一定能返回最新的数据或者最终版本的数据, 因为选择的这台服务器可能有一些更新延迟。

对于全局一致性的系统, 如果在同一时刻向两台服务器请求同一个数据, 那么得到的响应一定是相同的, 而对于一个最终一致性系统, 就未必了。最终一致性系统的每台服务器都会返回自己持有的数据, 而某台服务器持有的数据可能是旧版本的数据, 这个过期的数据也会返回给客户端。不过如果等待时间够长, 每台服务器持有的数据最终会变得一致, 都持有最新的数据。

图 5-23 所示为一个最终一致性场景。客户端 A 向最终一致性数据存储系统的服务器 1 发送一个更新操作。客户端 A 操作完成后 (即客户端 A 收到响应), 客户端 B 和客户端

C 立即向服务器 1 和服务器 2 分别查询这个数据。由于数据存储系统是最终一致性的，所以并不能保证不同服务器都得到最新的数据。客户端知道自己可以在某个时间点得到有效的数据，但是他们不知道现在得到的数据是否是最新的。在这个例子中，客户端 B 得到了最新的数据，而客户端 C 得到的是过期的数据，因为客户端 A 提交的更新还没有传递到服务器 2。如果客户端 C 等上一段时间再发送请求给服务器 2，就可以获得客户端 A 写入的数据了。

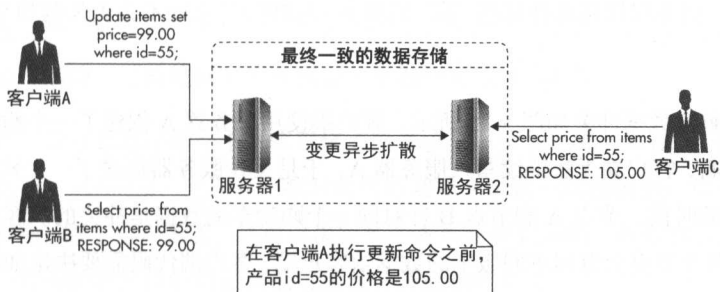


图 5-23 最终一致性

有些数据存储系统使用最终一致性的方案以提高系统可用性。客户端并不需要等待系统准备好以后再去执行读写操作。服务器可以在任何时候接受读写访问，然后将状态更新传递给对应的服务器。这种乐观写策略的消极影响是可能导致数据冲突，同一个数据可能在不同服务器上被同时更新。冲突如图 5-24 所示，当节点 A 和节点 B 在同步数据发现有冲突的时候，客户端 1 和客户端 2 已经完成操作去执行别的事务了。数据存储系统需要想办法处理 id=55 的数据达到一致性。

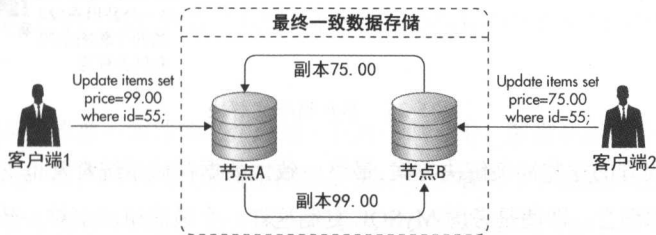


图 5-24 最终一致写冲突

解决冲突的办法有好几种，最简单的办法是接受最新的写入而丢弃前面的写入。这种方法被称为“最新写入赢”，这种方法虽然简单也很常用，但是可能会导致数据丢失。

还有一种方法，类似 Dynamo 这种，把解决冲突的责任交给客户端。系统检测冲突，并记录冲突的值。当有客户端访问有冲突的数据的时候，系统返回所有冲突版本的数据，由客户端决定如何解决冲突。客户端可以根据业务规则以一种比较优雅的方式解决各种类型的冲突。比如对于亚马逊的购物车，即使是有些服务器宕机了，用户还是可以向购物车添加商品。这些操作会写入不同服务器，每个服务器记录的购物车可能会有不同版本。当这些有不同版本的购物车被客户端代码发现的时候，客户端会把所有版本的购物车的商品都合并起来，而不仅仅是选择那些“赢”的版本。这样用户永远不会丢失购物车里的商品，购物更愉快。

客户端解决数据冲突如图 5-25 所示。客户端使用服务器 A 创建了一个购物车。由于网络临时中断，客户端无法继续写入服务器 A，于是又在服务器创建了一个购物车。当网络中断恢复的时候，节点 A 和节点 B 针对同一个购物车的数据是冲突的。客户端继续调用的时候，两个节点会返回不同版本的数据给客户端，客户端代码需要决定如何解决该冲突。在这个例子中，客户端决定将两个版本的购物车进行合并，并对服务器存储的购物车进行更新，数据存储系统的冲突消除了。

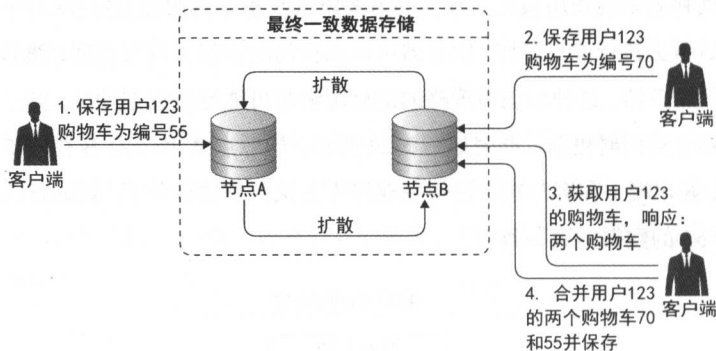


图 5-25 客户端冲突解决

除了前面提到的这些冲突解决方案，最终一致性数据存储系统常常也支持持续数据同步，以确保数据融合。即使是考虑 MySQL 复制这样一个最简单的最终一致性场景，只有一台服务器接受写操作，保证所有数据同步也是一件有挑战的事。小小的用户操作错误、应用 BUG、硬件异常也会引发从服务器和主服务器数据不一致。为了处理不同服务器存储的数据不一致，某些 NoSQL 数据存储，比如 Cassandra，会自身提供一些自愈策略。

例如, 10%的 Cassandra 读操作会触发一个后台读修复机制。在给客户端返回响应后, Cassandra 节点会从所有其他备份节点读取数据, 对这些值进行比较, 对不一致的或者过期的数据进行更新。虽然这种覆盖所有读操作的 10%的方式看起来有点多余, 但是要考虑到所有的节点都会接受写操作, 系统维护或者网络异常会导致数据不一致很容易扩散开来。数据自修复机制会带来额外的开销, 但是对于整个系统而言更容易从失效中恢复过来, 客户端可以在任何服务器上进行读写, 而不必寄希望于单台服务器的可用。

最终一致性是一种权衡, 选择最终一致性而不是全局一致性是一个艰难的决定。读到的数据可能是过期的, 写的数据有可能覆盖了不期望被覆盖的数据。

使用最终一致性数据存储并不意味着你不能在语义上实现“先写后读”。有些最终一致性系统, 比如 Cassandra, 允许客户端对每次查询独立设置一致性级别, 对一致性保证进行微调 and 权衡。除了全局配置, 还可以决定哪个查询需要更多的一致性保证, 而哪个查询允许接受一定的脏数据以获得更高的可用性并降低响应延迟。

投票一致性的意思是多数备份决定最终结果。使用投票一致性策略写数据的时候, 需要多数服务器确认数据已经持久存储。而使用投票策略进行读操作, 也就意味着多数服务器需要对客户端应答, 以便客户端可以从多个响应中根据投票策略找到最新的数据。

投票是最终一致性存储利用增加延迟解决一致性问题的办法。你需要更多的延迟时间以等待多数服务器返回响应, 但是可以获得最新的数据。如果你用投票一致性写入某个数据, 那么就可以用投票一致性读这个数据, 系统保证你可以得到最新的数据从而实现“先写后读”的语义。

图 5-26 所示为投票一致性的工作原理。在这个例子中, 数据在多个节点间复制。写入数据的时候, 至少写两个节点(在返回响应之前, 至少有两个节点确认已经将数据持久化存储)。也就意味着服务器 2 失效并不会影响系统的写入成功。稍后, 服务器 2 恢复运行重新提供服务, 客户端仍然可以获得最新的数据, 因为投票读可以读到其余两台服务器中至少一台, 里面就有最新的数据。

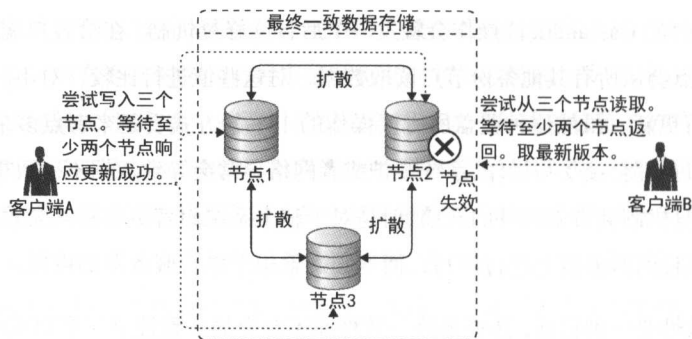


图 5-26 失效期间投票操作

快速恢复增加可用性

与 Dynamo 及 Cassandra 处理一致性和可用性策略一样，有些数据存储会牺牲一致性提高可用性，而有些数据存储设计者认为，与其保证所有的客户端在任何时间都能读写数据，不如关注如何让系统快速从失效中恢复过来，这样就不必牺牲全局一致性。

一个比较好的例子是 MongoDB，MongoDB 也是一个很流行的 NoSQL 数据存储。MongoDB 会将数据自动分片并存储在多台服务器中。数据的每一个分片属于某一台服务器，任何人想要更新数据都必须和这台服务器通信。也就意味着，任何时候如果这台服务器失效了，MongoDB 会拒绝这台服务器负责的这个分片的全部数据的写入。

用一台服务器负责一个分片有个显而易见的缺陷，任何时候如果有一台服务器失效，就会有一部分客户端操作失败。为了增加数据冗余提高可用性，MongoDB 支持复制集，系统推荐方案是为每一个分片设置一个复制集。在复制集中，多台服务器共享相同的数据，其中一台服务器被选举为主服务器。如果主节点失效了，则选举进程启动，在剩余的服务器中再选出一台服务器接任主节点的角色。一旦新的主节点选出来了，复制集的复制机制继续进行，新的主节点的数据会复制到其余的节点。利用这种方式，通过提高自动化提升失效恢复水平，可以使失效时间降到最短。

也许你觉得这个方案真的很赞——只需要在主节点失效的时候容忍一分钟的宕机时间，就可以获得一个一致性的数据存储。问题是每个应用的痛点是不一样的，事前无法预测应用对不可用的容忍程度。很多时候，数据存储系统的设计者们会以一种你难以预料的方式在各种约束中进行权衡。

再考虑一下 MongoDB 的一致性问题，事情其实比你想象的更复杂。你也许已经把 MongoDB 解读为一种 CP 数据存储（实现了一致性和分区耐受性），但是一致性的定义方式可能并不是你期望的那样。因为 MongoDB 的复制集使用的是异步复制，写操作先到达主节点，然后异步复制到副节点。这就意味着，如果主节点在数据复制到副节点之前失效，那么某些更新就永远丢失了。

主节点失效如何引起写丢失如图 5-27 所示。和 Cassandra 提高一致性类似，MongoDB 也可以在写操作的时候强制副节点和主节点保持一致。但是系统难道不是在缺省情况下就应该满足 CP 的吗？实践中，MongoDB 强制写操作同步复制到副节点的代价非常高，写操作复制并不是一个数据一个数据传递的，而是所有的复制日志都需要副节点刷出并处理的。

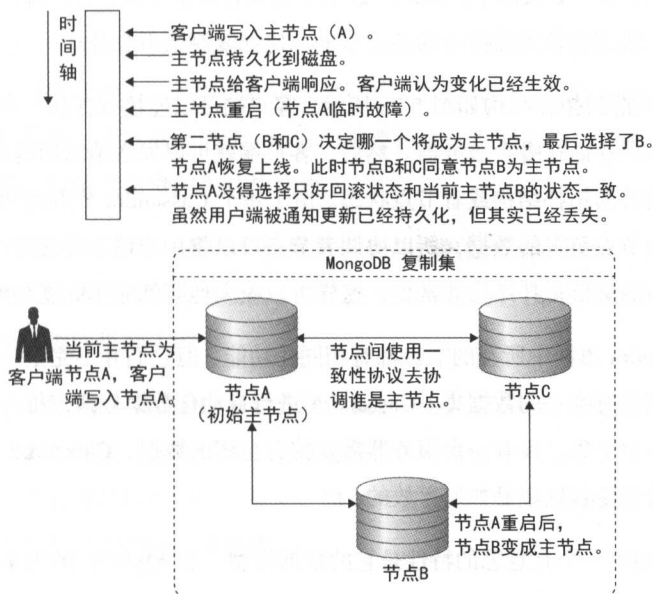


图 5-27 主节点失效导致更新丢失

很多现在的 NoSQL 数据存储支持自动失效转移或者失效恢复。无论选择哪一种 NoSQL，都需要对它进行深入研究，而不是胡乱猜测，实践中遇到的复杂性比文档上写的要复杂得多。

Cassandra 拓扑结构

NoSQL 数据存储的设计方法很多,有些非常有参考意义,在多台服务器上分布数据、复制信息、处理失效,这些方面不乏存在一些共同的模式。我们深入了解下 Cassandra,最流行的 NoSQL 数据存储之一,以及它的一些关键特性。

Cassandra 最早由 Facebook 开发,融合了一些 BigTable (Google 开发) 和 Dynamo (亚马逊开发) 的设计模式。

Cassandra 架构的一个主要特点是所有的节点在功能上都是相等的。Cassandra 不存在失效单点,所有的节点执行的功能都完全一样。客户端可以在需要的时候连接到 Cassandra 的任何一个节点上,这个节点就会变成这个客户端的会话协调者。客户端无须知道哪个节点上有哪些数据,也不必关注节点故障、数据恢复或者数据复制。客户端发送所有的请求给会话协调者,协调者负责集群内的各项事务,比如复制或者分片。

Cassandra 集群的拓扑结构如图 5-28 所示。客户端可以连接到任何一台服务器上,无论它想读写的数据存储在哪台服务器。然后,客户端就可以发送请求给这台协调者节点,无须知道整个集群的拓扑结构或者节点状态。由于每个 Cassandra 节点都知道所有其他节点的状态及这些节点负责的数据,所以协调者节点可以将用户请求分配给正确的服务器。客户端需要知道的集群拓扑结构非常少,这样可以极大地降低应用的复杂度和耦合度。

虽然 Cassandra 集群中所有的节点都有相同的功能,但它们并不完全一样。Cassandra 的每一个节点都持有唯一的数据集。Cassandra 进行自动化数据分区,每一个节点都得到整个数据集的一个子集。没有一台服务器需要持有全部的数据,Cassandra 节点之间互相通信,从而知道全部数据集是如何存放的。

Cassandra 的另一个有意思的特性是它的数据模型,和 MySQL 的关系数据模型非常不同。Cassandra 的数据模型基于宽表的结构,类似于 Google 的 BigTable^[w28]。宽表模型中可以创建表,每个表可以拥有无限的行。和关系模型不同,表之间没有关联,每张表都是独立的,Cassandra 不会对表和行之间进行任何强制约束。

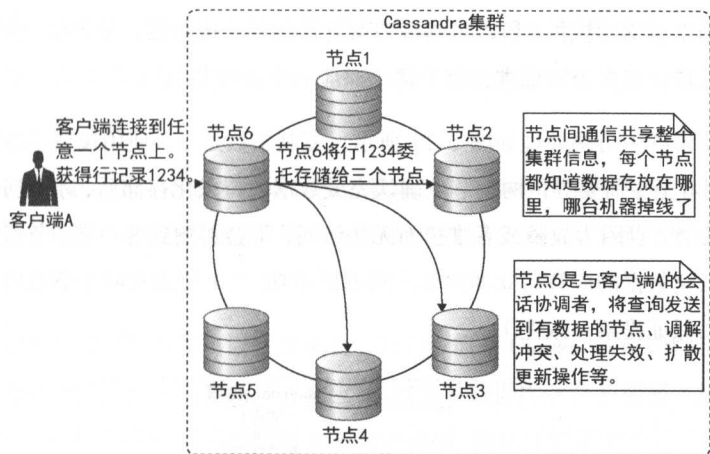


图 5-28 Cassandra 集群的拓扑结构

Cassandra 表定义也和关系数据库不同。不同的行可以有不同的列（字段），也可能存放在集群中的不同服务器上。无须事先定义表结构，可以在需要的时候动态创建字段。无须事先定义表结构是一个巨大的进步，当应用需要持久化存储新的信息类型的时候，可以快速变更而无须进行复杂昂贵的表结构变更。

Cassandra 数据模型简化的另一个作用是，当进行数据检索的时候，只需要更少的工具就可以了。为了检索任何列中的数据，需要先找到所在的行，为了找到行，需要知道行键（有点像关系数据库里的主键）。

类似本章早先提到的 MySQL 分片方案，Cassandra 基于行键进行数据分区。当发送请求给会话协调者的时候，会将行键（由应用提供）哈希成一个数字。然后，基于这个数据，可以找到这个行键对应的分区。最后，协调者查找哪个 Cassandra 服务器负责这个分区，并将请求发送给对应的服务器处理。

除了自动数据分区，Cassandra 也支持数据复制。需要注意的是，Cassandra 复制和 MySQL 复制不太一样。在 Cassandra 中，每个数据的备份都是同等重要的，服务器之间不存在主从关系。Cassandra 可以指定每个数据在整个集群中有多少个备份，会话协调者负责确保复制数量的正确。

写数据的时候，协调者节点将请求转发给负责相应分区的所有服务器。如果某台服务器宕机了，剩余的服务器仍然继续处理请求。发送给失效节点的请求会被缓冲下来，等服

务器恢复可用以后继续执行。所以虽然客户端连接到单个服务器，发送单个写请求，但是这个请求会被转化成多个写请求，每个请求对应一个备份节点。

写请求在集群中协调处理如图 5-29 所示，复制系数是 3，使用投票一致性。在这种场景下，协调者至少需要等待两个节点确认完成数据的持久化存储后，才会向客户端返回操作成功。某个节点因为故障或者维护而无法访问，不会影响到客户端，节点 6 会在收到两个节点持久化数据的应答后立即向客户端返回成功（三个节点有两个节点应答即满足多数原则，足以保证投票一致性）。

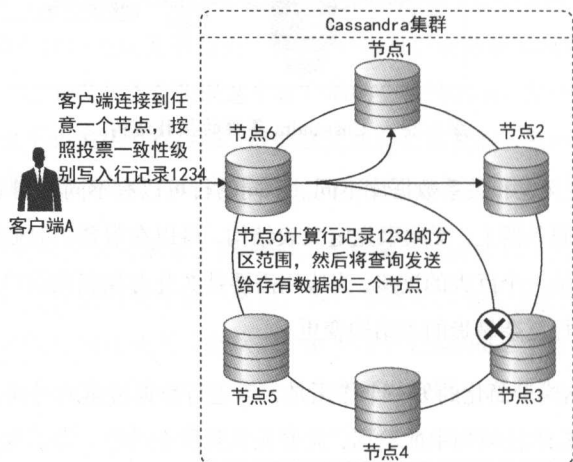


图 5-29 写入到 Cassandra

Cassandra 的另一个非常有意义的特性是高度自动化和最小化管理。例如，替换一个失效节点不用像 MySQL 那样进行复杂的备份恢复和复制补偿。替换宕机服务器只需要将一个空的（空白）服务器加入集群，然后通知 Cassandra 新节点要替换的 IP 地址是什么即可。所有的数据传输及一致性检查都在后台自动执行。由于每一份数据都存储在多台服务器上，所以整个集群所有服务器都会调动起来处理这个替换过程。客户端可以持续地读写数据，就像并没有一台服务器已经宕机并且正在被替换。当节点恢复完成，新节点就可以开始处理请求，集群的处理能力恢复到最初的状态。

从伸缩性角度看，Cassandra 是真正的水平伸缩数据存储系统。集群的服务器越多，读写处理能力越强，可以根据业务需求对系统进行伸缩。由于数据被切分到大量的小分区上，Cassandra 可以将数据均匀地分布到整个集群中。另外，Cassandra 向客户端隐藏了全

部的拓扑结构，可以相对自由地在服务器之间迁移数据。同样原理，新服务器可以加入集群作为一个新节点提供服务。另外，Cassandra 很注意集群的负载均衡，会确保新服务器能够分配到一定的数据。

可以看到，Cassandra 有诸多优点，易于管理，伸缩方便，具有自愈功能，但是任何事情都是有代价的。一个主要挑战就是 Cassandra 的使用方法非常特别，数据模型也很特别，此外就是最终一致性。

通过使用投票读写的方案可以实现最终一致性，但是 Cassandra 设计师设计的数据模型和各种权衡可能还是会让你感到不太适应。通过关系数据库学会的那套方法根本不适用于 Cassandra 这样的 NoSQL 系统。虽然大多数 NoSQL 数据存储系统很容易就能上手，但是想要自如地使用还是需要很多的经验，系统内部的结构也可能和期望的不太一样。

例如，虽然开源社区号称“Cassandra 擅长写操作”，但是 Cassandra 删除操作的代价非常巨大，使用者可能根本不会事先想到。大多数用户都不会期望删除操作代价昂贵，但这却是 Cassandra 开发者在各方面做出权衡后妥协的结果。Cassandra 使用一种只能向后追加的数据结构，以实现插入写操作的效率。存储在某个地方的数据永远不会被覆盖，磁盘永远不会执行随机写操作，这样极大地增加了写操作的吞吐能力。但是这个特性，连同 Cassandra 的最终一致性特性，使得删除与更新操作也不得不变成一个内部的持久性插入操作。导致的结果就是，某些应用添加和删除大量的数据会变得非常低效，因为删除也是在增加数据而不是减少数据（除非后台压缩进程将这些数据进行合并）。

以用 Cassandra 实现队列的反模式为例。可以用 Cassandra 动态列实现一个简单的先进先出队列。追加一个新列就是添加一个新条目，删除一个列就是从队列头移除一个条目。数据量不太大的时候，这个方法效果不错，但是随着不停地添加删除列，性能会急剧下降。虽然 Cassandra 会通过后台压缩机制将删除数据进行合并，但是如果删除操作的比例很高，效率会非常低下（这个例子中，50%的操作是删除操作）。

如果对 NoSQL 数据存储系统内部不了解，对其各种优点缺点不了然，就很容易在实践中掉到坑里。并不是说 NoSQL 不好，或者 Cassandra 不好，相反，某些情况下，NoSQL 是最好的选择，而 Cassandra 则是我最喜欢的 NoSQL 数据存储系统之一。这里想强调的是，虽然 NoSQL 支持水平伸缩，某些操作的性能极佳，但是 NoSQL 依然不是银弹，使用它需要付出代价。

小结

数据层的伸缩性通常是 Web 应用开发最有挑战的部分。通过仔细设计应用，选择合适的数据存储系统，灵活应用伸缩性三大基本技术：功能分割、复制及分片，可以实现数据层的水平伸缩。

无论使用哪一种数据存储，或者应用哪一种具体技术，一定要认清的是：数据存储设计是各种权衡的妥协。没有银弹，不同的应用需要的数据存储并不相同，需要的伸缩性也不一样。所以我强烈建议你打开思路，不要企图寻找一把完美的锤子。不要想着要找一个数据存储系统满足所有的应用需求，要认识到所有的系统都有优点也有缺点，所以尽量根据应用场景的不同混合使用各种存储系统。在 Web 服务层进行功能分割，基于业务场景使用不同数据存储系统，意图都是尽量支持多样化的持久存储方案，^[L37]这也是 Web 应用开发的一个发展趋势。虽然增加数据存储类型会增加应用复杂性及增加运维成本，但同时也带来更多灵活性，允许工程师针对每一种 Web 服务独立进行权衡选择，而不是把所有需求都强加到一个数据存储系统上。

在决定使用任何 NoSQL 数据存储系统之前，我建议你至少阅读一本关于数据存储的书，了解人们使用这种技术通常会遇到的问题和陷阱。如果想掌握更多数据存储伸缩性技术，我建议读一些更有挑战的书，关于 MySQL^[16]和 MongoDB^[44]的书，描述不同 NoSQL 数据存储有一些非常不错的白皮书。^[w28, w29, w27, w18, w72, w55]

不论数据存储系统设计得如何好，应用设计得如何好，I/O 仍然是大多数系统的瓶颈。第 6 章我们将讨论缓存，这是减少数据层负载最简单的一种方法。

6

缓存

“战争的最高境界是不战而屈人之兵。”

——孙子

缓存是伸缩 Web 应用的一项关键技术，在用低成本提升性能和伸缩性上起着重要作用。缓存技术相对简单，很容易添加到现有应用中而不需要复杂的架构技术。缓存在很多方面可以做到不战而屈人之兵，即不用做实际计算就能响应请求，这使得伸缩更容易。缓存技术的简单性使得它非常流行，其在许多技术中都被采用也证明了它的成功，包括 CPU 内存缓存，硬盘缓存，Linux 操作系统文件缓存，数据库查询缓存，域名服务系统（DNS）客户端缓存，超文本传输协议（HTTP）浏览器缓存，HTTP 代理和反向代理等各种类型的应用对象缓存。这些场景中，缓存被用来减少访问时间和消耗的资源。缓存技术将处理结果保存在缓存区，而不用每次请求都从源头获取数据再生成结果。后续请求直接从缓存获取结果并返回，直到数据过期再被删除。由于所有缓存对象都能从源头构建，因此缓存可以在任何时间点被清除或丢失，而不会导致任何不良后果。如果缓存对象无法找到，那么重新构造它即可。

缓存命中率

缓存命中率是运用缓存技术时简单而重要的一项指标。本质上，缓存是否有效依赖于

你能多少次重用同一个缓存响应业务请求，这个度量指标被称作缓存命中率。如果一个缓存结果能平均满足 10 个请求，那么它的命中率是 90%，因为每个对象仅需要生成一次而不是 10 次。有三个重要因素影响缓存命中率：缓存键集合大小、内存空间和缓存寿命。接下来，我们仔细看看每个因素。

影响缓存命中率的第一个因素是缓存键集合大小。缓存中的每个对象使用缓存键进行识别，定位一个对象的唯一方式就是对缓存键执行精确匹配。例如，如果想为每个商品缓存在线商品信息，你需要使用商品 ID 作为缓存键。换句话说，缓存键空间是你的应用能够生成的所有键的数量。从统计数字上看，应用生成的唯一键越多，重用的机会越小。例如，如果想基于客户 IP 地址缓存天气数据，则可能有多达 40 亿个键（这是所有可能的 IP 地址的数量）。如果要基于客户来源国家缓存天气数据，则可能仅需几百个缓存键（世界上所有国家的数量）。一定要想办法减少可能的缓存键数量。键数量越少，缓存的效率越高。

影响缓存的第二个因素是用缓存可使用内存空间的大小。这直接决定了缓存对象的平均大小和缓存对象数量。因为缓存通常存储在内存中，缓存对象可用空间受到严格限制且相对昂贵。如果想缓存更多的对象，就需要先删除老的对象，再添加新的对象。替换（清除）对象会降低缓存命中率，因为缓存对象被删除后，将来的请求就无法命中了。物理上能缓存的对象越多，缓存命中率就越高。

影响缓存的第三个因素是平均每个对象在过期或失效前在缓存中的生存时间，这个时间称为 TTL。在某些场景中，例如，缓存天气预报数据 15 分钟没问题。在这个场景下，你可以设置缓存对象预定义 TTL 为 15 分钟。在其他场景中，你可能不能冒险使用过于陈旧的数据。例如，在一个电子商务系统中，店铺管理员可能在任何时刻修改商品价格，这些价格需要准确地展示在整个网站中。在这个场景下，你需要在每次商品价格修改时让缓存失效。简单讲，对象缓存的时间越长，缓存对象被重用的可能性就越高。

理解这些基本因素是有效使用缓存的关键。读写比高的数据是使用缓存不错的场景，因为缓存对象只被创建一次，然后在过期或失效前都可以长期保存使用；而数据更新频率极高的场景下使用缓存的意义就不大，因为缓存中的对象在重用前往往已经失效了。

在后面的部分，我们会讨论 Web 应用相关的两种主要缓存：基于 HTTP 的缓存和定制对象缓存。然后我会介绍每个场景使用的技术和有效利用缓存的一些通用规则。

基于 HTTP 的缓存

HTTP 协议已经存在很长时间了,这些年一些新的扩展不断添加进 HTTP 规范,允许 Web 基础设施的不同组件可以缓存 HTTP 响应。这使得对 HTTP 缓存的理解更困难,你会发现许多不同的 HTTP 头都与缓存相关,甚至一些 HTML 元标记也与缓存相关。我们会介绍关键的 HTTP 缓存头,但在深入细节之前,有一个重要概念需要提及,就是所有 HTTP 层的缓存都是通读缓存。

通读缓存是一个缓存组件,它能给客户端返回缓存资源,或在请求未命中缓存时获取实际数据(例如当缓存不包含请求的对象时)。这意味着客户端连接的是通读缓存而不是生成响应的原始服务器。

图 6-1 所示为客户端、通读缓存和原始服务器之间的交互。缓存通常作为中介(或称为代理),透明地给 HTTP 连接添加缓存功能。在图 6-1 中,客户端 1 连接到缓存并请求一个特定的 Web 资源(一个页面或层叠样式表 CSS 文件)。然后缓存拦截请求并用缓存中的对象给客户端返回响应。只有当缓存没有合适的缓存数据进行响应时,才会连接到原始服务器,并转发客户端请求给它。因为服务和通读缓存的接口是一样的,所以客户端也可以直接连接到服务(如客户端 2)从而跳过缓存,通读缓存只有在被连接上时才会工作。

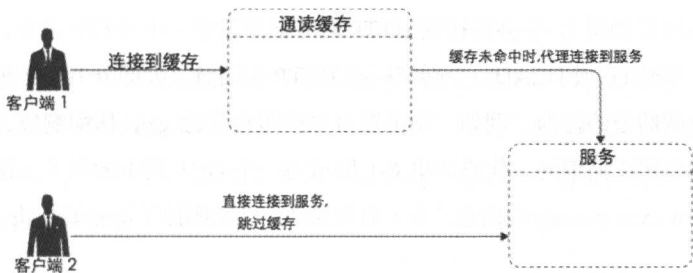


图 6-1 客户端与通读缓存交互

通读缓存很有吸引力,因为它对客户端来说是透明的。客户端无法区分收到的对象是否是被缓存的。如果客户端直接连接到原始服务器,它就可以跳过缓存并获取一样正确的响应。这种可拔插架构带来了很大的灵活性,使得你可能给 HTTP 栈添加若干个缓存层而不用修改任何客户端。实际中,确实可以看到多个 HTTP 通读缓存存在一个 Web 请求中互

相链接起来，而客户端却感知不到它们。

图 6-2 所示为串联 HTTP 通读缓存的样子。本章后面会更详细地介绍这些缓存，现在只需要注意客户端连接可以被多个通读缓存拦截而客户端不会感知到它们就可以了。每一步中，代理使用缓存的响应内容发回给请求方，或在缓存未命中时将请求转发给源头。接下来，我们详细看看怎样用 HTTP 协议头来控制缓存。

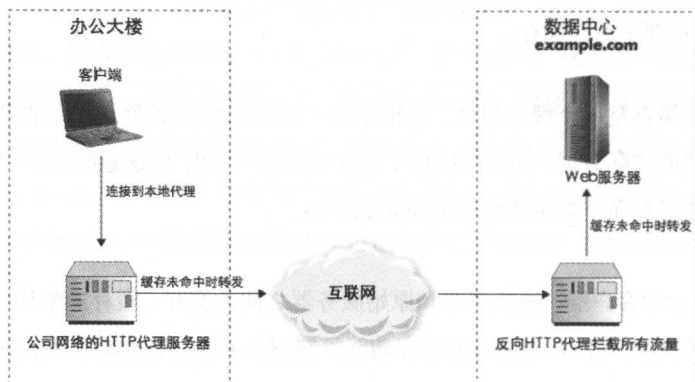


图 6-2 HTTP 通读缓存链

HTTP 缓存头

HTTP 是一个基于文本的协议。当浏览器请求一个页面或其他 HTTP 资源时（图片、CSS 文件或 AJAX 调用），它会连接到 HTTP 服务器并发送一个 HTTP 命令，例如 GET、POST、PUT、DELETE 或 HEAD，并附带一些 HTTP 头信息。大部分 HTTP 头信息是可选的，被用以协调期望的行为。例如，浏览器可以声明它支持 gzip 压缩响应，这可以让服务器决定是否返回压缩编码。代码清单 6-1 所示为一个 GET 请求的例子，这个请求用来获取 <http://www.example.org/> 的信息。为了简化说明，已经删掉了不必要的头，例如 cookie 和 user-agent。

代码清单 6-1 GET 请求的例子

```
GET / HTTP/1.1
Host: www.example.org
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

这个例子中，浏览器声明它支持 HTTP 1.1 版本的协议及压缩数据编码。它也告诉 Web 服务器要查找哪个主机和 URL。最后，它请求 Web 服务器让指定的 TCP/IP 连接保持打开状态，这样它就可以发送更多的 HTTP 请求继续下载资源，例如图片、CSS 文件，以及 JavaScript 文件。

在给客户端的响应中，Web 服务器会应答类似代码清单 6-2 所示的数据。注意服务器决定使用 gzip 压缩编码算法返回响应，因为这是客户端建议的，但它拒绝了保持长连接的要求（keep-alive 头），返回一个响应头（connection:close）并立即关闭连接。

代码清单 6-2 简单的 HTTP 响应的例子

```
HTTP/1.1 200 OK
Content-Encoding: gzip
Content-Type: text/html; charset=UTF-8
Content-Length: 9381
Connection: close
```

... response body with contents of the page ...

这些是客户端和服务端会用到的不同的请求头和响应头，但我只会关注与缓存相关的头。缓存头实际上很难恰当使用，因为有很多选项可以设置。更复杂的是，一些较老的头比如“Pragma:no-cache”在不同的实现下解释也不同。

提示

你可以使用相同的头来控制 Web 页面和静态资源的缓存，比如图片和 Web 服务响应等。缓存 Web 服务响应的能力实际上是 REST-ful 服务的关键扩展伸缩能力。通常，你可以在 Web 服务和客户端之间放置 HTTP 缓存，并利用与 Web 页面缓存相同的机制。

你需要熟悉的第一个头是 Cache-Control。Cache-Control 被添加到 HTTP 1.1 规范中，现在得到大部分浏览器和缓存包的支持。Cache-Control 允许你指定多个选项，如代码清单 6-3 所示。

代码清单 6-3 HTTP 头 Cache-Control 的例子

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

Web 服务器包含在响应中的最重要的 Cache-Control 选项有：

- private

指示请求结果对请求用户是特定的，响应不能提供给其他用户。在这个用法中，意味着只有浏览器可以缓存响应，因为中间缓存不知道怎样识别一个用户。

- public

只要它未过期，指示响应可以在用户间被共享。注意，你不能同时指定 private 和 public 选项；响应要么是 public 的要么是 private 的。

- no-store

指示响应不能被任何中间缓存存储到磁盘上。换句话说，响应可以被缓存在内存，但它不能被持久化到磁盘。你应该在响应包含用户敏感信息时使用包含这个选项，这样浏览器或中间缓存都不会存储这些数据到磁盘上。

- no-cache

指示响应不能被缓存。更精确地说，它表达的意思是缓存需要询问服务器此请求在每次用户请求相同资源的时候是否仍然合法。

- max-age

指示响应在缓存失效前，客户端可以保持缓存多少秒（定义了响应的 TTL）。由于潜在的 inconsistency，这个信息可以用几种方式表达。我推荐不要用 max-age（它向后兼容不好），而且它还依赖 Expires 这个 HTTP 头。

- no-transform

指示响应不做任何修改直接用缓存提供。例如，一个内容分发网络（CDN）提供商可能通过对图片转码来减少大小，降低质量或改变压缩算法。

- Must-revalidate

指示一旦响应过于陈旧，不重新验证就不能返回给客户端。尽管看起来有点奇怪，但在特定条件下缓存确实有可能返回过期对象，例如客户端允许这么做，或者缓存失去了与

原始服务器的连接。通过使用 `must-revalidate`，你告诉缓存必须停止提供陈旧响应数据。每当客户端请求陈旧对象，缓存会强制向原始服务器请求数据。

注意只要未超过过期时间，就认为缓存对象是“新鲜的”。一旦超过过期时间，对象就变为“陈旧的”，但如果客户端显式地允许陈旧响应，仍可以把它返回给客户端。如果你想禁止陈旧对象返回给客户端，就需要在 `Cache-Control` 响应头中包含 `must-revalidate` 选项。客户端也能在请求中包含 `Cache-Control` 头。`Cache-Control` 头很少被客户端使用，当包含在请求头中时，其语义有轻微差异。例如，包含在请求中的 `max-age` 选项告诉缓存，客户端不接受比 `max-age` 更旧的对象，即便这些对象仍然被缓存认为是“新鲜的”。

另一个与缓存相关的 HTTP 头是 `Expires`，它指定一个缓存对象失效的绝对时间点。代码清单 6-4 所示为怎样使用 `Expires` 头的一个例子。

代码清单 6-4 HTTP-Expires 的例子

```
Expires: Sat, 23 Jul 2015 13:14:28 GMT
```

不幸的是，正如你看到的，一些 `Cache-Control` 头控制的功能覆盖了其他 HTTP 头的功能。Web 响应的过期时间可以由 `Cache-Control:max-age=600` 定义，也可以使用 `Expires` 头定义一个绝对过期时间。在响应中同时包含这两种头是冗余的，会导致混乱和潜在的不一致行为。因此，推荐使用你想用的头并固定使用它们，而不是将所有可能的头都用在响应中。

另一个重要的头是 `Vary`。这个头的目的是告诉缓存你需要基于某些 HTTP 请求头，生成响应的多个变体。代表清单 6-5 所示为最常用的 `Vary` 头，指示需要基于客户端发送给服务器的 `Accept-Encoding` 头用不同的方式对响应进行编码。某些客户端接收 `gzip` 编码的压缩响应，而其他不支持 `gzip` 头的则会得到未压缩的响应。

代码清单 6-5 HTTP 头 Vary 的例子

```
Vary: Accept-Encoding
```

还有一些 HTTP 头与缓存相关，允许你有条件地下载资源和重验证，但这些信息超出了本书的范围。这些头包括 `Age`、`Last-Modified`、`IF-Modified-Since`，以及 `Etag`，这些会另外研究。接下来我们来看一些常用的缓存场景的例子。

第一个场景允许你的客户端永久缓存响应。这是一项非常重要的技术，你可以应用到所有静态内容（比如图片、CSS 或 JavaScript 文件）上。静态内容文件必须不可变，当你需要变更这种文件时，最好用一个新的 URL 重新发布它。例如，当你部署一个 Web 应用的新版本时，可以将所有 CSS 文件捆绑并缩小，然后包含一个时间戳或文件内容的 hash 码在 URL 中，例如：`http://example.org/files/css/css_a8dbcf212c59dad68dd5e9786d6f6b8a.css`。

提示

捆绑 CSS 和 JavaScript 文件并用一个 URL 发布可以带来两个好处：静态文件可以被缓存永久缓存（在浏览器、代理服务器、CDN 服务器）；在任何时间点同一个文件会有多个版本。这能将缓存命中率最大化并让新代码更容易部署。如果你通过替换已有的 URL 部署 JavaScript 文件的新版本，那么一些 HTML 页面版本较旧的客户端会加载新的 JavaScript 文件并产生错误。通过用新的 URL 来发布新版本静态文件，你可以确保用户总是能下载到 HTML、CSS、JavaScript 的兼容版本。

即便你可以永久缓存静态文件，你也不能设置 Expires 头超过 1 年（HTTP 规范不允许设置超过这个期限）。代码清单 6-6 所示为一个静态文件 HTTP 头的例子，允许缓存至多 1 年（从 2014 年 7 月 23 日开始计时）。这个例子也允许同一个对象的缓存在不同用户间重用；确保压缩、非压缩对象独立于缓存，防止产生编码错误。

代码清单 6-6 HTTP 头静态文件的例子

```
Cache-Control: public, no-transform
Expires: Sat, 23 Jul 2015 13:14:28 GMT
Vary: Accept-Encoding
```

第二个常见的场景是一个对缓存而言最糟糕的案例——此场景下你想要 HTTP 响应不被存储、缓存或被任何用户重用。为此，你可以像代码清单 6-7 所示那样使用响应头。注意，这里我使用另一个 HTTP 头（`Pragma:no-cache`）来确保老的客户端能理解我不缓存响应的意图。

代码清单 6-7 HTTP 头不缓存内容的例子

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Pragma: no-cache
```

最后一个场景是你希望同一个用户可以重用内容,但同时不想让其他用户共享缓存的响应。例如,如果网站允许用户登录,你可以将用户资料名展示在页面右上角,并显示一个链接指向资料页。在这个场景中,页面内容包含用户特定数据,因此你不能在不同用户间重用响应。你仍然可以使用这个页面的缓存,但这必须是私有缓存,以便确保用户仅能看见他们自己的私人页面。代码清单 6-8 所示为 HTTP 头允许 Web 浏览器将这种响应缓存一段时间(例如,从 2015 年 7 月 23 日,13:04:28 GMT 开始 10 分钟)的例子。

代码清单 6-8 HTTP 头允许 Web 浏览器将响应缓存的例子

```
Cache-Control: private, must-revalidate
Expires: Sat, 23 Jul 2015 13:14:28 GMT
Vary: Accept-Encoding
```

最后一个需要注意的地方是,除了 HTTP 缓存头,你会发现还有某些 HTML 元标记看起来也能控制 Web 页面缓存。代码清单 6-9 所示为一些元标记。

代码清单 6-9 要避免的缓存相关的 HTML 元标记

```
<meta http-equiv="cache-control" content="max-age=0" />
<meta http-equiv="cache-control" content="no-cache" />
<meta http-equiv="expires" content="Tue, 01 Jan 1990 1:00:00 GMT" />
<meta http-equiv="pragma" content="no-cache" />
```

最好避免使用这些元标记,因为它们不能用来控制中间缓存,而且容易引起混乱,尤其对于一些经验不足的工程师来说。最好还是用 HTTP 头来控制缓存,并且用最小的冗余来做。现在我们已经讨论了 HTTP 缓存是怎样实现的,接下来我们看看可以用来提升 Web 网站性能和伸缩性的不同类型。

HTTP 缓存技术类型

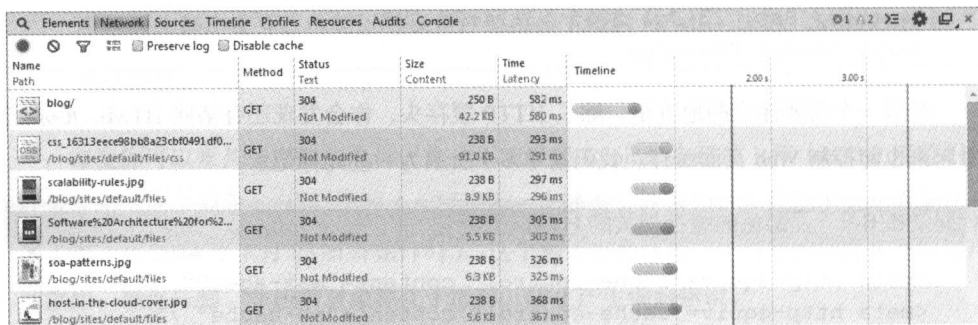
HTTP 协议在实施 Web 客户端和服务端间的缓存时提供了很大的灵活性。有许多种使用 HTTP 缓存的方式,并且可以相对容易地将它们插入已有的系统中。目前,主要有 4

种 HTTP 缓存：浏览器缓存、缓存代理、反向代理和 CDN。其中大部分都不需要自己进行扩展，因为它们由用户设备或第三方网络控制。本章将讨论基于 HTTP 缓存的伸缩性，但首先我们会讨论每种 HTTP 缓存。

浏览器缓存

第一种最常见的缓存是在 Web 浏览器中嵌入的缓存层，称为浏览器缓存。浏览器的内建缓存能力可以减少发出的请求次数。这些通常与内存和本地文件结合使用。每当 HTTP 请求将要发出，浏览器会检查缓存是否有该资源的合法版本。如果响应存在于缓存中并仍然有效，浏览器就可以重用它而不用发出一个 HTTP 请求。

图 6-3 所示为 Google Chrome 浏览器开发者工具栏的一部分。第一次页面加载时，在当前下载的 Web 资源序列中，可以看到下载 HTML 页面的时间是 582ms，之后 CSS 文件连同几张图片需要下载，每个都需要花费 300ms。



Name Path	Method	Status Text	Size Content	Time Latency	Timeline
blog/	GET	304 Not Modified	250 B 42.2 KB	582 ms 580 ms	
css_16313eece98b8a23cbf0491df0... /blog/sites/default/files/css	GET	304 Not Modified	238 B 91.0 KB	293 ms 291 ms	
scalability-rules.jpg /blog/sites/default/files	GET	304 Not Modified	238 B 8.9 KB	297 ms 296 ms	
Software%20Architecture%20for%20... /blog/sites/default/files	GET	304 Not Modified	238 B 5.5 KB	305 ms 303 ms	
soa-patterns.jpg /blog/sites/default/files	GET	304 Not Modified	238 B 6.3 KB	326 ms 325 ms	
host-in-the-cloud-cover.jpg /blog/sites/default/files	GET	304 Not Modified	238 B 5.6 KB	368 ms 367 ms	

图 6-3 首次访问下载资源序列

如果返回的 HTTP 头允许浏览器缓存这些响应，就能有效提升页面加载速度并节省服务器的大量渲染、发送工作。图 6-4 所示为同样的页面加载顺序，但这次大部分资源直接由浏览器缓存提供。虽然页面自身还需要较长时间验证，但所有图片和 CSS 文件都可以由缓存提供而没有任何网络延迟。

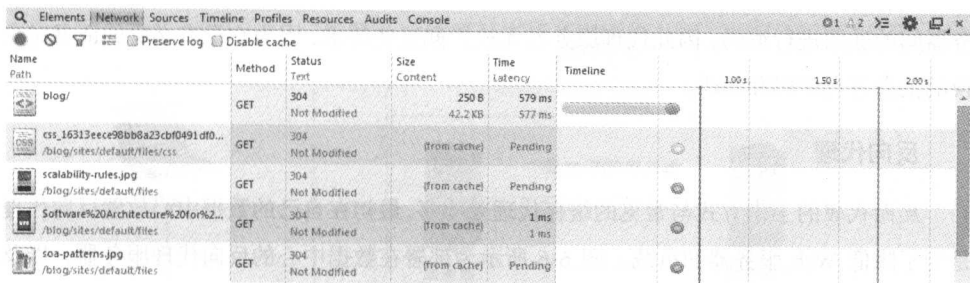


图 6-4 连续访问下载资源序列

缓存代理

第二种 HTTP 缓存技术是缓存代理。缓存代理是一个服务器，通常安装在本地公司网络或互联网服务商处（ISP）。它是一个通读缓存，用来减少网络用户产生的流量，方式是在网络用户间重用响应。网络越大，节省的潜力也就越大，这也就是为什么所有 ISP 都喜欢使用该技术的原因。ISP 安装透明的缓存代理并尽可能多地将 HTTP 请求路由到缓存中。图 6-5 所示为透明缓存代理是怎样安装在本地网络中的。

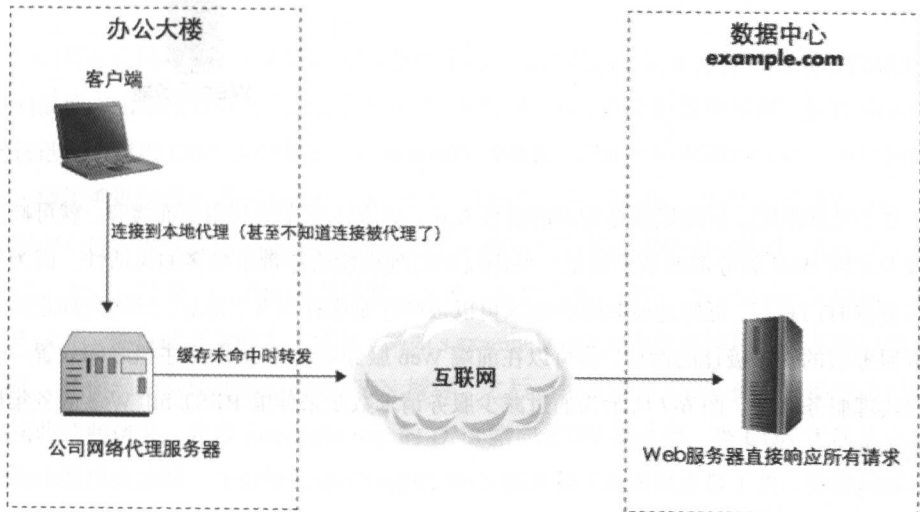


图 6-5 本地网络中的 HTTP 代理服务器

近年来，安装本地代理服务器的做法已经不那么流行了，一则因为带宽更便宜，二则因为 Web 站点越来越多地采用安全套接字层（SSL）提供资源服务。SSL 对客户端和服

务器间的通信进行编码,因此代理服务器无法拦截这些请求,因为它没有必要的证书对交换的消息进行解码和编码。

反向代理

反向代理的工作方式与常见的缓存代理差不多,最初在自己的数据中心放置反向代理是为了降低 Web 服务器的负载。图 6-6 所示为部署在数据中心的反向代理服务器,以及 Web 服务器,从 Web 服务器返回的响应被反向代理缓存。

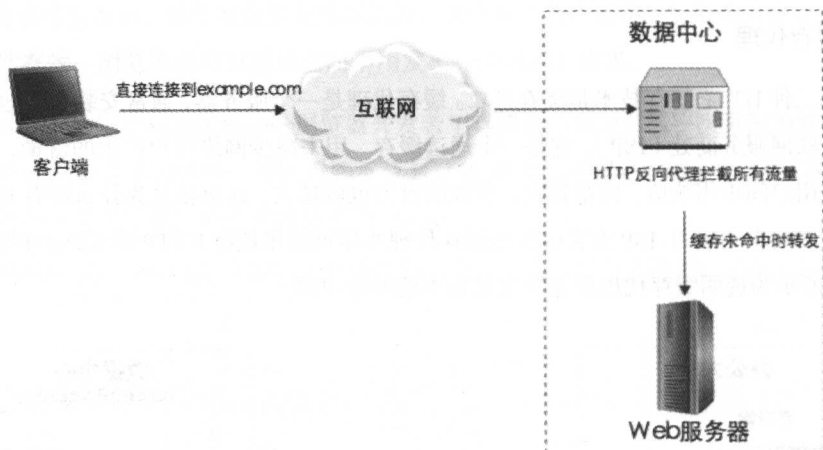


图 6-6 反向代理

对于某些应用,反向代理是伸缩的极佳方式。如果你能充分利用页面缓存,就可以大幅减少发到 Web 服务器的请求数量。使用反向代理也能给你带来更多的灵活性,因为你可以覆盖 HTTP 头,更好地控制哪些请求可以被缓存及缓存多久。最后,反向代理是加速 Web 服务层的一个极佳的途径。你可以在前端 Web 服务器和 Web 服务主机之间放置一个反向代理服务器层。图 6-7 所示为通过减少服务请求数量来伸缩 REST-ful Web 服务集群的示意图。

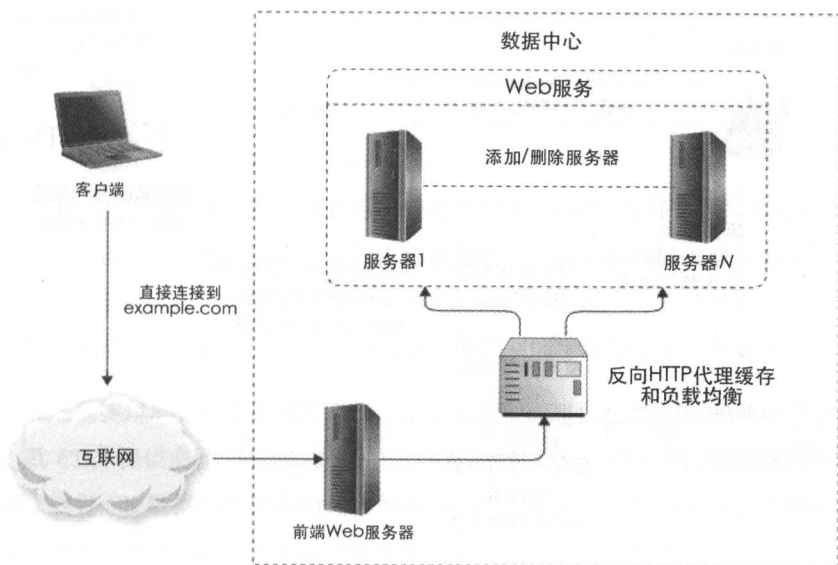


图 6-7 Web 服务集群前置反向代理

内容分发网络

内容分发网络是一个由缓存服务器构成的分布式网络，其工作方式与缓存代理类似。它们依赖于相同的 HTTP 头，但是内容分发网络由 CDN 服务提供商控制。由于 Web 应用增长迅速，使用 CDN 变得非常有价值。利用 CDN 可以减少服务器的负载，节省网络带宽，使用更近的服务器将内容推送给用户，从而提升用户体验。CDN 提供商通常有许多数据中心位于世界各地，这使他们能从最近的缓存服务器中提供缓存结果，因此降低网络延迟。Web 应用使用 CDN 来缓存静态文件，例如图片、CSS、JavaScript 和视频，或者 PDF 文档。你可以提供静态子域名（例如 s.example.org）并为所有使用此域名的资源生成 URL，从而轻松实现静态资源的 CDN 缓存。之后，你可以在 CDN 提供商那里进行配置来接收这些请求，并将 s.example.org 的 DNS 指向 CDN 提供商。当 CDN 无法从自己的缓存中提供内容时，它会转发请求到你的 Web 服务器（原始服务器）上，并为后续用户缓存这些响应。图 6-8 所示为 CDN 怎样用来缓存静态文件。

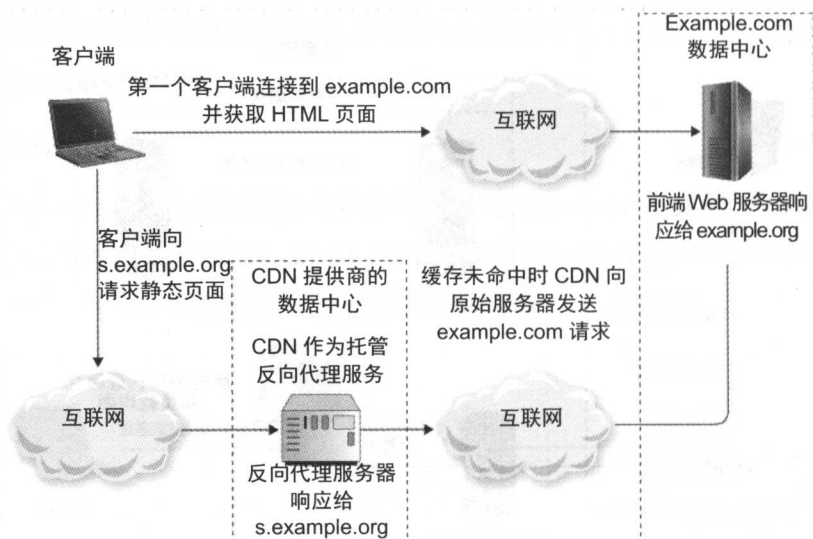


图 6-8 CDN 配置静态文件

某些 CDN 提供商可以通过配置，同时提供网站的静态和动态内容，这样客户端就不用直接连接到数据中心，动态内容请求会穿透 CDN 提供商的缓存服务器。这项技术有一些好处。例如，提供商可以减轻分布式拒绝服务攻击（就像 CloudFlare 提供的），也能进一步减少发送到原始服务器的 Web 请求，因为动态内容（即便是私人内容）也可以被 CDN 缓存。图 6-9 所示为怎样配置 Amazon CloudFront 来同时处理静态和动态资源。

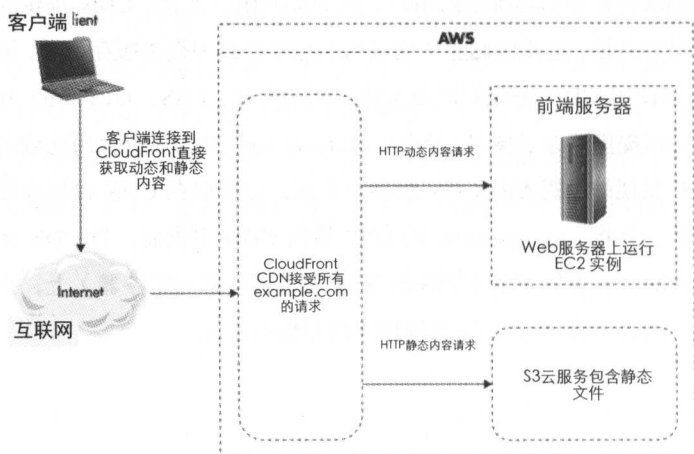


图 6-9 CDN 同时配置静态文件和动态内容

现在,我们已经看过了不同类型的 HTTP 缓存。接下来,我们看看随着网站流量增长怎样伸缩每种缓存。

伸缩 HTTP 缓存

基于 HTTP 的缓存技术如此受欢迎的一个原因是:你可以将自己的服务器负载推到离用户更近的第三方管理的机器上。任何由浏览器缓存,第三方缓存代理或 CDN 提供服务的请求,都不会进入你的服务器,最终降低基础设施的负载。同时,HTTP 缓存服务的请求比你自己的服务器响应得更快,这使得基于 HTTP 的缓存技术显得额外有价值。

正如之前提过的,不要担心浏览器缓存或第三方代理服务器的伸缩性,它们不受你的控制。就 CDN 提供商而言,你也不用担心其伸缩性,因为 CDN 可以透明地伸缩,按每百万请求或每 GB 数据来收取你的固定费用。通常,每单位价格随着你的扩容伸缩而下降,使得成本更低。你只需要管理反向代理服务器就可以了,如果使用反向代理的话,你需要自己对其进行管理和伸缩。

市面上有许多开源的反向代理服务器,包括 Nginx、Varnish、Squid、Apache mod_proxy,以及 Apache Traffic 服务器。如果你托管在一个私有数据中心,你也可以使用某些硬件负载均衡器提供的内置反向代理服务器功能。

对大部分年轻的初创企业来说,单个反向代理服务器应该足够处理所有的入口流量了,因为硬件反向代理服务器和领先的开源产品(Nginx 或 Varnish)都能单机处理每秒超过 10,000 次的请求量。因此,通常情况下,决定缓存哪些数据,缓存多久,比怎样伸缩反向代理服务器更重要。

要想有效地伸缩反向代理层,首先需要关注缓存命中率。同样,受本章开始提到的三个因素的影响,具体到反向代理,它们可以表述为以下内容。

- **缓存键空间**

它描述了反向代理在一段时间内(比如 1 小时)需要记录多少唯一 URL 的数量。唯一 URL 数量越多,需要每台反向代理服务器的内存或存储就越多,以便能够通过缓存提供相当比例的流量服务。要避免按用户缓存响应(例如,URL 中包含用户 ID),这种类型的响应很容易出现大量无法重用的对象占满缓存的情况。

- **平均响应存活时间 TTL**

它描述了响应被缓存的时间。缓存时间越长，重用的机会就越大。尽量永久缓存对象。如果不能永远缓存对象，则要与利益相关方协调一个可接受的最长缓存 TTL。

● 缓存对象的平均大小

它影响了反向代理能够使用多大的内存或存储，来保存最常访问的对象。缓存对象的平均大小是最难控制的，但你仍需要关注它，目前有一些技术能帮助你缩减对象大小。例如，CSS 文件和 JavaScript 文件能被缩小；HTML 可以进行预处理，删除在模板渲染阶段产生的多余的空格符和注释。

值得指出的是你不必担心设置一个较长的 TTL 会填满缓存服务器，因为内存缓存设计了相应的算法来清除很少访问的对象并回收空间。最常用的算法是最近最少使用(LRU)算法，它允许缓存服务器删除极少访问的对象，并在内存中保存热点数据以最大化缓存命中率。

一旦你确认哪些是需要尽可能长时间缓存的对象，以及哪些是能有效重用缓存的对象，你就可以开始考虑伸缩反向代理层。你需要尽可能地让反向代理达到并发极限或吞吐量极限。这些问题可以通过并行部署多台反向代理及分发流量来解决。

图 6-10 所示为一个部署场景，此处使用了两层反向代理服务器。第一层反向代理直接部署在负载均衡器后面，以便在反向代理服务器之间分发流量。第二层反向代理位于前端服务器和 Web 服务主机之间，在这个例子中，前端服务器设置为每个请求随机获取一个反向代理服务器。等到你的技术栈规模增长得更大的时候，就需要在前端服务器和反向代理服务器之间部署一个负载均衡器，以使配置变得更透明和独立。幸运的是，你不需要这样复杂的部署，通常前端应用之前的反向代理是不需要的，把它推到 CDN 更合适。

提示

如果你直接使用 HTTP 缓存，添加更多反向代理并行运行，通常不会引起什么问题。HTTP 协议不需要在 HTTP 缓存间保持同步，且并不确保所有的客户端请求通过同一个物理网络进行路由。每个 HTTP 请求可以在一个单独的 TCP/IP 连接中发送并被不同的中间缓存路由。客户端必须在这些约束下工作，接收不一致响应或采用缓存重验证。

无论你采用什么反向代理技术,你都可以采用相同的多代理平行部署模式,因为其底层行为几乎是相同的。每个代理都是一个独立的备份,和其他机器之间不共享数据,这就是为什么在考虑伸缩性时反向代理技术的选择没有那么重要的原因。通常情况下,推荐使用 Nginx 或硬件反向代理,因为它们有极佳的性能和功能特性。一些值得提的 Nginx 特性包括:

- Nginx 仅使用异步处理,这让它用非常低的单连接开销就能代理 10 万级的并发连接。
- Nginx 也是一个 FastCGI 服务器,意思是你可以在同一个 Web 服务器栈上运行 Web 应用,同时又作为你的反向代理。
- Nginx 能扮演负载均衡器的角色,它支持多种转发算法及许多高级特性,例如 SPDY、WebSockets 及节流。Nginx 也能在配置后改写 HTTP 头,这可以用来给没有实现缓存头的 Web 应用提供 HTTP 缓存,或者改写它们的缓存策略。
- Nginx 有一个活跃的社区。据报道,在 2013 年 Nginx 为超过 15% 的互联网应用提供服务。

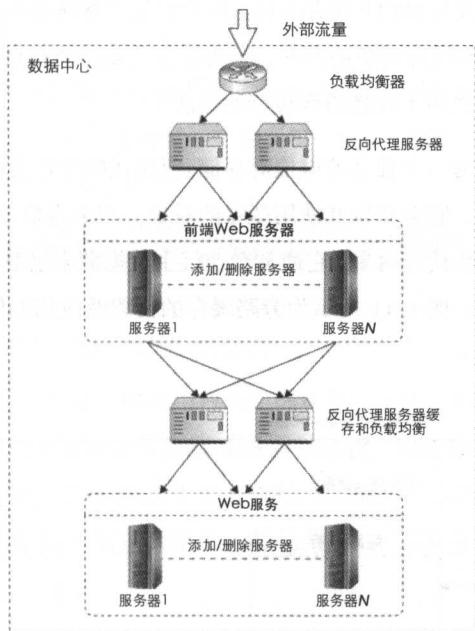


图 6-10 多层反向代理服务器

如果你托管数据中心并有一个硬件负载均衡器，我推荐你将它作为一个反向代理使用，这样可以减少技术栈中的组件数量。在其他场景中，我推荐你研究现有的开源反向代理服务器，比如 Nginx、Varnish 或 Apache Traffic 服务器，选择其中一个，并通过增加副本进行水平伸缩。

最后，你也可以通过提供更多的内存或将持久化存储更换为固态硬盘（SSD）来对反向代理进行垂直伸缩。SSD 技术在缓存对象占用内存比缓存服务器工作内存大很多的情况下尤其有用。为了提高缓存命中率，你可以通过使用文件存储将缓存容量提升为上百 GB，而不是仅仅依靠共享内存。相对于传统（自旋）硬盘，SSD 的随机访问速度更快，通过使用 SSD 硬盘，能将响应时间提升至少 10 倍。同时，由于缓存数据是可以被丢弃的，所以你也不用担心 SSD 的寿命限制或突然的断电失效（与 SSD 崩溃有关）。

缓存应用对象

探讨过基于 HTTP 的缓存之后，Web 应用栈中第二个重要的缓存组件是定制对象缓存。对象缓存的使用方式与 HTTP 缓存不同，因为它是旁路缓存而不是通读缓存。在旁路缓存中，应用需要意识到缓存对象的存在，旁路缓存主动存储和获取对象，缓存并不是透明地处于应用和数据源之间（这是通读缓存的做法）。

旁路缓存被应用视为一个独立的键值对存储。应用代码通常会询问对象缓存需要的对象是否存在，如果存在，它会获取并使用缓存的对象。如果需要的对象不存在或已过期，应用会做必要的工作从头构建对象，它通常会连接主数据源来组装对象，并将其保存回对象缓存中以便将来使用。图 6-11 所示为旁路缓存的位置及应用如何直接与数据源通信而不是通过缓存通信。

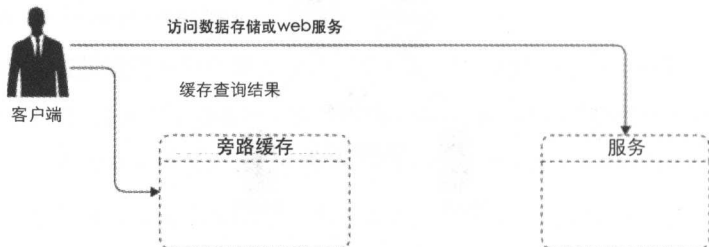


图 6-11 旁路缓存

与其他类型的缓存类似,使用对象缓存的主要动因是节省从头构建应用对象的时间和资源。这部分讨论的所有缓存类型可以被想象为支持过期时间的键值对存储,并且它们支持简化的编程接口,允许你基于唯一的键字符串来获取、设置和删除对象。接下来,我们近距离看看不同类型的对象缓存及它们的优缺点。

对象缓存的一般类型

就像本章早些时候讨论的 HTTP 缓存一样,对象缓存的部署方式也有多种。实际采用的技术取决于使用的技术栈,但概念是相似的。

客户端缓存

首先,我们看看直接位于客户端设备的本地存储。几年前,JavaScript 还不能在客户端机器上存储信息,但现在,大多数 Web 浏览器支持 Web 存储规范,允许 JavaScript 直接在用户设备上存储应用数据。即便 Web 存储允许应用存储的空间有限(通常 5MB ~ 25MB),这仍然是一种非常好的提升 Web 应用速度的方式,同时还减轻了对后端基础设施的压力。其更有价值的地方在于,可以很容易地存储特定用户的信息,因为 Web 存储在单个设备上隔离的。

代码清单 6-10 所示为在 Web 存储中存储对象是多么容易。Web 存储工作模式类似键值存储。要存储一个对象,你需要提供一个称为键(key)的唯一标识,以及你要持久化的字符串字节,这称为值(value)。

代码清单 6-10 在 Web Storage 中存储对象的 JavaScript 代码

```
var preferences = { /* data object to be stored */ };
localStorage.setItem('preferences', JSON.stringify(preferences));
```

Web 存储永久地存储对象,即使用户关闭浏览器后重启计算机,对象依然可以被访问。用户可以通过设置浏览器配置来清除 Web 存储数据,因此你需要记住 Web 存储仅仅是缓存而不是可靠的存储。当你想访问 Web 存储中的值时,你需要使用与持久化时相同的键来请求它。代码清单 6-11 所示为用 JavaScript 如何从 Web 存储中获取对象。

代码清单 6-11 访问持久对象的 JavaScript 代码

```
var cachedData = localStorage.getItem('preferences');  
var preferences = JSON.parse(cachedData);
```

在开发单页面应用时（SPA），本地设备存储显得更为重要，因为这些应用在浏览器中需要运行更多的代码并执行更多的异步 Web 请求（AJAX）。特别地，如果你为移动设备开发 SPA（例如，使用 Sencha Touch 框架）应用，你需要集成本地存储方案来缓存 Web 服务响应并减少需要发送到 Web 服务器的请求数量。与 Web 存储类似，在开发本地移动应用时，你可以直接在客户端设备上存储数据。在这种情况下，技术不同，但操作原理是相似的。当使用本地设备存储时，需要记住，它与数据中心是分离的，这使得服务器不能直接对缓存对象进行删除或失效操作。每当你使用 Web 存储或类似的客户端存储时，你需要在前端 JavaScript 代码中包含负责缓存刷新及失效的代码。举个例子，假定你在开发一个移动设备 SPA，让用户查看当前位置步行距离内的餐厅。由于是移动应用，你可能想加速应用加载时间或通过本地存储减少数据传输。你可以在应用打开时给用户展示上次的搜索结果，而不是在等待定位（GPS）和搜索结果时展示一个空白屏幕。

为此，你需要在 Web 存储中保存每次的搜索结果，以及用户坐标和时间戳；然后在加载的时候，你可以先从 Web 存储中加载，仅仅展示上次的搜索结果。同时，后台程序将当前位置和时间与上次搜索进行比较。如果用户位置时间显著改变——我们假定移动超过 200 米或已经 1 天没有打开过应用——你就需要更新用户界面，向用户显示更新正在进行，然后发送一个异步请求给服务器来加载新数据。这样，用户就能立即看到一些东西，这使得应用看起来响应很快。同时，也减少了发送给服务器的不必要的请求，以防用户去餐馆的路上多次打开应用。

本地缓存

另一类对象缓存是直接位于服务器上的缓存。无论你开发一个前端 Web 应用还是一个 Web 服务，都能从本地缓存中获益。本地缓存通常使用以下几种方式实现。

- 对象直接缓存在应用内存。应用创建一个缓存对象池并不再释放分配的内存。在这个场景下，访问缓存对象时实际上没有开销，因为它们以执行代码的格式直接存储在进程管理的内存中。不需要复制、传输或对象编码，可以被直接访问。这种方式适用于所有编程语言。

- 对象存储在共享内存，同一台机器的多个进程可以访问它们。这种方式的开销仍然很小，共享内存方法与本地进程内存几乎一样快。这种实现会增加一些开销，但可以认为影响微乎其微。例如，在 PHP 中，在共享内存中存储对象会迫使对象序列化，这增加了一点开销，却可以使同一台机器运行的所有进程都能够访问共享缓存对象池。不过这种方式并不太常用，特别在像 Java 这样的多线程环境中不太适用，其所有执行线程都运行在一个进程中。
- 缓存服务器作为独立应用部署在 Web 服务器上。在这个场景中，每个 Web 服务器都有一个运行在本地的缓存服务器实例，但仍需要使用缓存服务器接口与缓存交互而不是直接访问共享内存。这种方式在小型 Web 应用程序中更常见，开始只是单台服务器，并将缓存（Memcached 或 Redis）与 Web 应用安装在同一台机器上，主要可以节省托管成本和网络延迟。这种方式的好处是应用以和外部缓存服务器交互的方式使用缓存——实际上，缓存却运行在同一台机器上，这使得当需要将缓存部署到一个专用集群中时，几乎不需要修改任何代码。

以上每种方法最后都可以归结为相同的概念，即将对象缓存在应用代码执行的本地机器上。这样的好处是持久化和访问速度都很好。由于对象存储在同一台机器上，这样的访问可以比从远程服务器获取快几个数量级。表 6-1 所示对使用本地内存、磁盘，以及远程网络调用的延迟时间。

表 6-1 访问不同资源的延时情况

操作类型	粗略时间
访问本地内存	100 ns
SSD 磁盘搜索	100,000 ns
网络数据包在同一数据中心来回一次的时间	500,000 ns
磁盘搜索（非-SSD）	10,000,000 ns
按顺序从网络读取 1MB 数据	10,000,000 ns
按顺序从磁盘（非-SSD）读取 1MB 数据	30,000,000 ns
跨大西洋网络数据包一次来回延时	150,000,000 ns
每秒等于多少	1,000,000,000 ns

使用本地应用缓存的一个额外好处是开发和部署的简单性。不用部署额外组件，以及运维管理，不用进行服务器间通信，本地缓存仅仅可以看作是应用进程分配的额外内存。

本地缓存也不需要服务器间进行同步和复制,不用担心锁和网络延迟,这使得事情更简单,速度更快。通过在每台服务器上建立唯一的和独立的本地缓存,可以使 Web 集群通过增加副本就能方便地实现伸缩(如第 2 章和第 3 章描述),因为这样的 Web 服务器是可互换且彼此独立的。

不幸的是,本地缓存也有一些缺点。最重要的一点是,每台应用服务器最后会缓存相同的对象,导致服务器间很多数据重复。这是由于应用服务器上的缓存没有任何信息共享,它们也不进行任何同步。如果每台机器分配 1GB 内存给对象缓存,不论有多少台机器,最后你的整个集群也只能缓存 1GB 的数据,每台 Web 服务器都会受到这个 1GB 的限制,而所有的服务器都缓存相同的 1GB 数据。根据场景的不同,这个限制可能很关键,因为你不可能很容易地伸缩缓存的大小。

本地缓存的另一个重要限制是缓存无法保持一致,你不能快速地从所有的本地缓存中删除对象。例如,假定你在构建一个电子商务系统,你要缓存产品信息,那么你可能需要在产品价格变化的时候删除这些缓存对象。不幸的是,如果在多台没有任何同步和协作的机器上都缓存了该对象,除非构建非常复杂的方案,否则不可能立即从这些缓存中删除对象(比如发布消息到 Web 服务器上,要求从缓存中删除特定的对象)。

分布式对象缓存

与 Web 应用相关的最后一种常见对象缓存是分布式对象缓存。这种缓存与本地缓存主要的差别是:与分布式缓存交互需要与缓存服务器进行网络通信。从好的方面说,分布式缓存提供了比本地缓存更好的伸缩性。分布式缓存通常也以键值对的方式工作,允许客户端将数据存储在分布式缓存中一段时间,这之后对象会自动从缓存服务器删除。目前已经有许多开源的产品,例如在 Web 领域非常受欢迎的 Redis 和 Memcached。此外,有许多商用产品也可以考虑,比如 Terracotta 服务器阵列和 Oracle Coherence,不过我推荐创业公司使用开源产品解决方案。

与分布式缓存服务器交互非常简单,大部分缓存服务器都有常见编程语言的客户端程序库。代码清单 6-12 所示为一个简化的缓存接口。你需要指定想要连接的服务器,以及要存储的键和值,以及生存时间 TTL(秒),这个时间之后对象会从缓存中删除。

代码清单 6-12 PHP 代码将用户数在 Memcached 中缓存 5 分钟

```
$m = new Memcached();  
$m->addServer('10.0.0.1', 11211); //set cache server IP  
$m->set('userCount', 123, 600); // set data
```

在远程缓存服务器（例如 Redis 或 Memcached）缓存对象有一些好处，最重要的是可以更好地对其进行伸缩。关于伸缩性我们会在后面详细讨论，现在，假定你仅能简单地向缓存集群添加服务器，通过添加服务器，可以同时伸缩缓存吞吐量和整体内存池大小。使用分布式缓存，你还可以有效地从缓存中删除对象，以便在源数据变化时让缓存失效。之前已经解释过，在一些场景中，一旦数据发生变化，你需要立即从缓存中删除对象。使用分布式缓存使得处理缓存失效更容易，你需要做的仅仅是连接到缓存，然后发送删除对象命令即可。

使用专用缓存也是将缓存职责从应用中移除的一个不错的方法，因为缓存服务器只是数据存储，并且需要支持很多不同的特性。例如，Redis 允许数据持久化、复制，以及实现分布式计数器、列表和对象集。谈到缓存管理，缓存也可以进行深度优化，它们可以处理对象过期和清理。

提示

当到达内存限制时，缓存服务器通常采用 LRU 算法来决定要从缓存中删除哪些对象。每当你要在缓存中存储一个新对象时，缓存服务器会检查是否有足够的内存。如果没有剩余空间，最近最少使用的对象就会被删除以保证有足够空间来存储新对象。通过使用 LRU 缓存，你不用担心从缓存中删除数据——它们或是过期被删除，或是为了给更“热”的数据腾出空间。

分布式缓存通常部署在一个独立的服务器集群上，并安装比一般服务器更多的内存。图 6-12 所示为缓存服务器是如何部署的——从前端服务器到 Web 服务主机都能访问的独立缓存服务器集群。

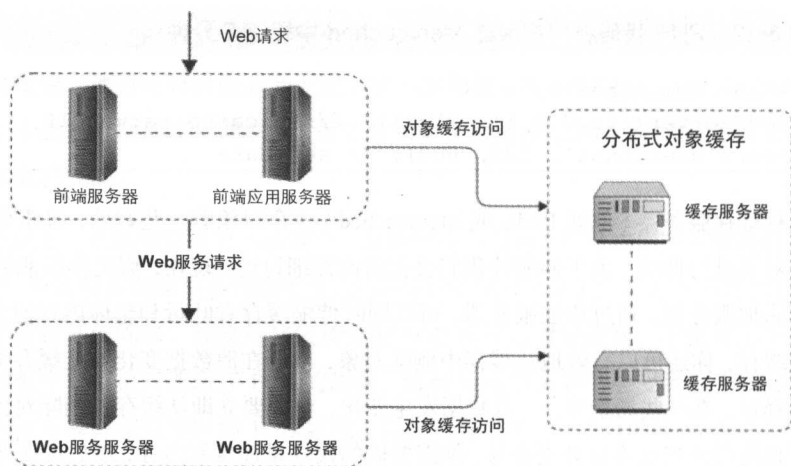


图 6-12 分布式对象缓存部署

虽然分布式缓存是一个强大的伸缩工具，并且在结构上相对简单，但在系统中仍然会增加一定的复杂度和管理开销。如果你使用云托管 Redis 和 Memcached 缓存，那么你不需要担心部署和服务器管理，但仍需要理解和监控它们，以保证有效地使用它们。当部署新缓存时，开始时规模可以尽可能小一些，Redis 是一个高效的缓存服务器，单机能支持每秒几十万次操作，这让你无须伸缩就可以支撑相当大的业务流量。一旦吞吐量不是问题，就可以通过添加更多内存进行垂直伸缩，而不是采用复制和数据分区等复杂部署方案。不过，随着系统增长流量更大，就需要进行水平伸缩了。现在我们进一步看看如何伸缩对象缓存。

伸缩对象缓存

当要进行对象缓存伸缩时，选择的方案取决于缓存的位置和类型。例如，客户端缓存比如浏览器缓存就无法进行伸缩，因为无法改变浏览器允许你使用的内存容量。Web 存储的特点是用户使用的是自己的缓存，你可以持续增加用户，无须伸缩客户端缓存以存储用户特定的数据。

Web 服务器本地缓存的伸缩通常需要用文件系统，因为没有其他方式来分发或增加单机上的缓存。在某些场景中，你可能有一个非常大的数据池，那里每个对象可以被缓存很长时间但却很少访问。在这样的场景下，更好的方式是使用 Web 服务器的文件系统以序列化方式存储缓存对象，而不是存储在共享缓存集群的内存中。虽然访问缓存在文件系统上的对象很慢，但不需要远程连接，因此 Web 服务器更为独立，可以与其他子系统

的失效保持隔离。基于文件的缓存相对更便宜，因为磁盘存储比操作内存更便宜，而且你不必为共享对象缓存创建独立的集群。随着 SSD 硬盘越来越流行，基于文件系统的缓存可能会成为内存缓存又便宜又快的替代品。

再说到分布式对象缓存，基于采用的不同技术，可以使用不同的方式进行伸缩，但通常数据分区（第 2 章和第 5 章有解释）是最好的方式，因为你可以伸缩集群的吞吐能力和整体内存池的大小。一些技术，例如 Oracle Coherence 本身集成了数据分区功能，但大部分开源解决方案（例如 Redis 和 Memcached）所采用的客户端数据分区的方式则更为简单。

如果你决定采用 Memcached 作为对象缓存，那么情况就非常简单了。你可以使用 Memcached 客户端库的内建特性实现将数据分区存储在多台服务器上。不用自己写代码实现，你可以直接告诉客户端库你有多台 Memcached 服务器。代码清单 6-13 所示为使用本地 PHP 客户端声明一个 Memcached 集群的多台服务器是多么简单，它底层使用了 libMemcached 与 Memcached 服务器通信。

代码清单 6-13 多台 Memcached 服务器作为一个集群

```
<?php
$cache = new Memcached();
$cache->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE, true);
$cache->addServers(array(
    array('cache1.example.com', 11211),
    array('cache2.example.com', 11211),
    array('cache3.example.com', 11211)
));
```

通过声明 Memcached 集群，数据会使用一致性 hash 算法自动在缓存服务器间分布。当你发送一个 GET 或 SET 命令时，Memcached 客户端库会对要访问的缓存键计算 hash 值，并将其映射到其中一台服务器上。一旦客户端找到指定键对应的服务器，就将请求只发送给那台服务器，这样集群的其他服务器就不用参与操作了。这是一个无共享方式的例子，因为每个缓存对象都只被分配给一台服务器而不会在多个服务器间产生冗余和协调。

图 6-13 所示为一致性 hash 是如何实现的。首先，所有可能的键呈现为一个范围的数字，首尾连接创建一个环。然后将所有服务器放在这个环上，彼此间隔相等，然后声明每个服务器负责存储的缓存键介于它和下一个服务器之间（沿着环顺时针移动）。这样，知

道了缓存键和集群的服务器数量，就可以找到哪台服务器有你要找的数据。

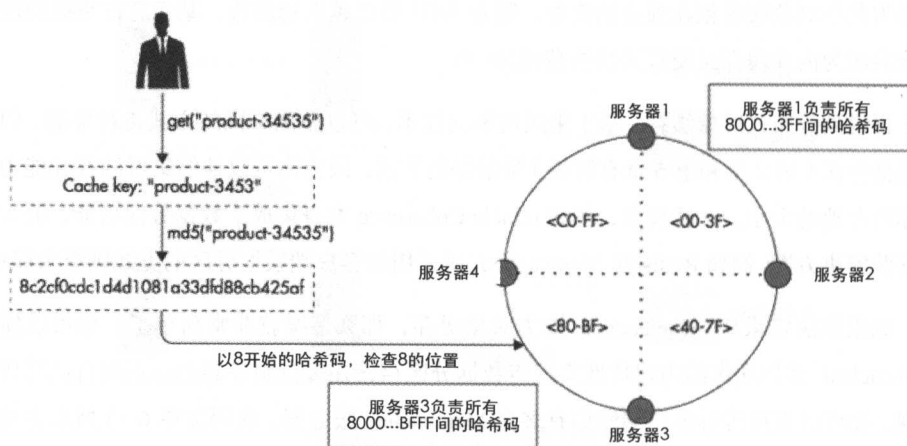


图 6-13 基于一致性哈希的缓存分区

要对缓存集群进行水平伸缩，需要能够向集群中添加服务器，这正是一致性 hash 的亮点。既然每台服务器只负责环上键空间的一部分，那么添加一台新的服务器到集群上会导致每台服务器轻微移动。这样，只有一小部分缓存键子集会在服务器间重新分配，这引起的缓存不命中的波动相对较小。图 6-14 所示为从 4 台机器的集群伸缩到 5 台机器的集群时服务器位置是怎样变化的。

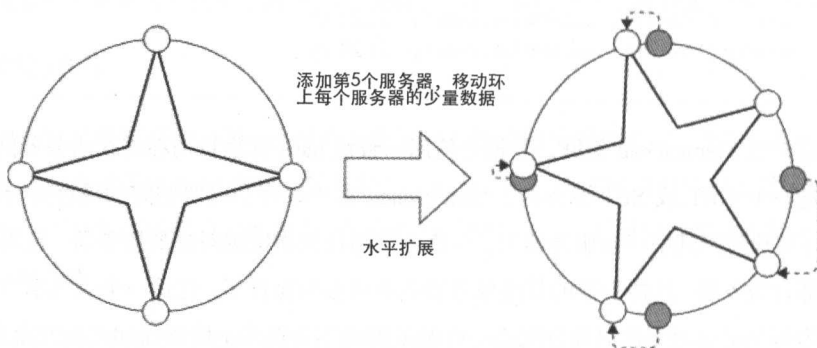


图 6-14 基于一致性哈希进行缓存集群扩展

如果你使用了朴素分区方案，例如使用取模函数来映射缓存键到服务器编号，每当在集群中添加或删除服务器，大部分缓存键会重新分配，导致极大地清洗整个缓存，即大部

分缓存对象不能命中。PHP 的 Memcached 客户端不是唯一支持一致性哈希的客户端库。事实上，如果你的缓存驱动不支持现成的一致性哈希算法，还有许多开源库可以选择。

提示

为了更好地理解缓存，可以将缓存看做一个大的哈希表。缓存能快速定位数据的原因就是哈希函数能判断缓存对象应该放在哪个“桶”里。这样，无论缓存多大，Get 和 Set 操作都能以常数时间执行。

另一个可选的伸缩对象缓存的方法是使用数据复制，或者混合使用数据分区和数据复制。一些对象缓存，比如 Redis，允许主从复制部署，这在一些场景中很有用。例如，如果一个缓存键变得很“热”，那么所有应用服务器都需要同时并发获取，这种情况就可以采用读副本。所有需要访问该缓存对象的客户端不必连接到一台单独的服务器，可以通过添加这个节点的只读副本来伸缩集群（见第 2 章）。图 6-15 所示为怎样部署每台缓存服务器的只读副本伸缩读吞吐量并支持更高的并发。

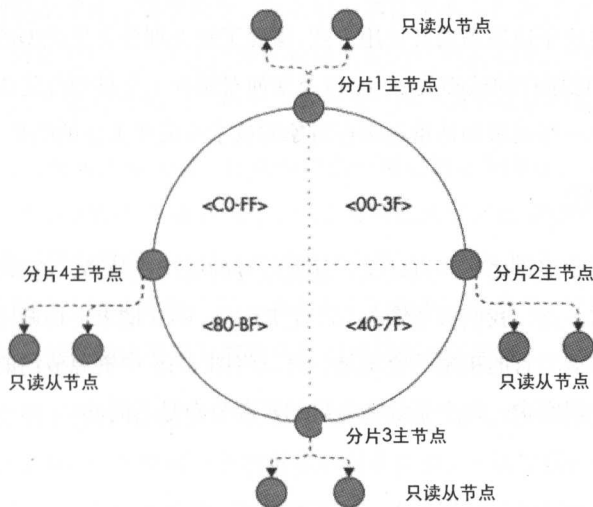


图 6-15 基于数据分区和复制扩展缓存集群

需要指出的是，如果之前你将 Web 应用托管在 Amazon 上，那么你可以选择将自己的缓存服务器部署在 EC2 实例上，也可以选择使用 Amazon Elastic Cache。不幸的是，Elastic Cache 可能不如你期望的那么聪明，因为本质上它只是一个托管的缓存集群，其主要的价

值在于你不用管理服务器或者担心失效恢复的场景。当你创建了一个 Elastic Cache 集群,你可以选择使用 Memcached 还是 Redis,也可以选择需要多少台服务器及每台服务器的容量。要记住,你需要自己在客户端代码中实现缓存服务器间的负载分发,因为 Elastic Cache 不能添加透明分区或者自动伸缩。类似地,你也可以选择其他云托管提供商来创建缓存集群。Azure 允许你部署一个托管 Redis 实例,只需简单几次单击就可以实现复制和自动容错。

对象缓存总体来说比数据存储更容易伸缩,使用简单的数据分区或复制就足以进行集群水平伸缩。思考一下,所有存储在对象缓存中的数据都是可以丢弃的,那么一致性和持久性约束就可以很宽松,从而允许更简单的伸缩。现在我们已经讨论了不同类型的缓存及它们的伸缩性技巧,接下来,我们继续讨论一些在设计可伸缩 Web 应用时有用的通用经验法则。

缓存的经验法则

缓存的难度取决于应用需求及使用方式,需要了解大部分常见类型的缓存及它们的伸缩方式。我们会讨论缓存的核心关注点,以及如何对缓存工作排列优先级以便获取最大收益。我们也会讨论一些能帮助你重用缓存对象的技术及需要关注的陷阱。我们开始吧。

缓存整个调用栈

关于缓存,其中最重要的一点就是:你能缓存的调用栈越高,能节省的资源就越多。为了更好地说明这一点,我们参考图 6-16,它展示了 Web 请求平均调用栈的样子,以及通过缓存每一层你能节省的粗略的资源量。可以将图 6-16 中资源节约的百分比看做一个简单的经验法则。现实中,每个系统在每层的资源分布是不同的。



图 6-16 技术栈各层的缓存

首先，客户端请求一个页面或资源。如果资源在其中一个 HTTP 缓存中可用（浏览器或本地代理）或者能从本地存储中得到，那么服务器甚至不会看到这个请求，这样就能节约 100% 的资源。如果失败了，次优选择是直接反向代理或 CDN 上获取 HTTP 请求，在这种场景下，为了生成整个响应，你也只需要承受两个百分点的资源消耗作为代价。

当一个请求发送到 Web 服务器，你仍有机会使用定制对象缓存，不调用 Web 服务就提供整个响应。如果你需要调用 Web 服务，你也可能从部署在 Web 应用和 Web 服务之间的反向代理服务器获得响应。只有这个机制也失败了，Web 服务才会参与处理请求。这里再次强调，你能使用对象缓存并完成请求而无须数据存储参与。只有所有这些都失败了，你才需要查询数据存储或搜索引擎来获取用户需要的数据。

这个原则也适用于你的应用代码。相比于仅仅缓存用来构建页面的数据库查询结果，如果你缓存整个页面片段，显然可以节省更多时间及资源。正如你看到的，缓存的终极目标是避免 Web 请求到达 Web 服务器，即便做不到，你仍需要尽量高地缓存整个调用栈。

用户间缓存重用

使用缓存的另一个要点是：尽可能多地在不同请求或用户间重用相同的缓存对象。缓存对象如果不被再次请求，那么就是时间和资源的浪费。

我们考虑下面这个例子。想象一下你在构建一个移动应用，允许用户查找当前位置附近的餐厅。主用例是用户看到步行距离内的餐厅列表，这样他们就可以选择喜欢的餐厅并快速找到吃的。应用的简单实现是检查用户 GPS 坐标，构建一个包含用户当前位置的查询字符串，访问应用服务器请求附近餐厅列表。Web 编程接口（API）的请求类似代码清单 6-14 所示。

代码清单 6-14 请求经度：151.209146，纬度：-33.880381

```
GET /restaurants/search?lat=-33.880381&lon=151.209146
```

这个方法的问题是几乎每个请求的参数都不同。即便只走几步也会改变 GPS 的位置，使得 URL 都不相同，导致缓存完全无用。

针对这个问题的一个较好的解决方案是将 GPS 位置设置为三位小数，这样同一街区的每个人都可以重用搜索结果。不用记录城市范围内几十亿可能的位置，就可以有效地减少可能位置的数量，并增加响应的缓存命中率。由于 URL 不包含用户特定数据且无须个性化，那么就没有理由不重用整个 HTTP 响应。

假设你在悉尼提供餐厅服务，并且决定使用三位小数的经纬度，那么你就可以将用户位置的可能数据量降低至 100 万以内。既然只有 100 万可能的响应，那么就能将这些响应缓存在反向代理层中（甚至 CDN 中）。因为餐厅细节信息不可能立即变化，响应结果即使缓存几个小时也不会对业务造成任何影响，还能进一步增加缓存命中率。代码清单 6-15 所示为如何保持 URL 不变，仅对请求参数做出变化，就大大地减少了可能被请求的 URL 数量。

代码清单 6-15 请求经度：151.207，纬度：33.867

```
GET /restaurants/search?lat=-33.867&lon=151.207
```

在多个用户间重用相同数据的原则适合很多场景。你应该寻找一些方法，通过这些方法可以多次返回相同对象而不用从头生成它们。如果不能缓存整个页面，也可以缓存页面片段，或使用其他技巧来减少可能的缓存键的数量（就像那个餐厅查找的例子一样）。总之，关键点在于你需要最大化缓存命中率，要做到这一点，你可以通过增加缓存池，延长对象缓存 TTL，减少可能的缓存键的数量来实现。

从哪儿开始使用缓存？

如果你在优化一个没有充分使用缓存的 Web 应用，你需要问自己，从哪儿开始下手？哪些查询是重要且需要缓存的？哪些页面值得缓存？哪些服务需要的缓存最多？因为就任何优化来说，都需要基于一个严格而简单的度量标准进行优先级排列，而不是仅仅依靠直觉。要评估哪些地方需要缓存，可以用生成特定响应的聚合时间来度量。聚合时间可以采用下列方式计算：

$$\text{聚合时间} = \text{每个请求消耗的时间} \times \text{请求数量}$$

这种方法能让你确定缓存哪些页面（或资源）是最有价值的。例如，之前我工作过的一家 Web 网站流量相当大，我们打算同时改善伸缩性和提升性能，因此我们开始尝试比较激进的缓存用法。要决定从哪下手，可以通过 Google Analytics 报告的前 20 个页面的平均响应时间等相关统计数据来判断。之后我基于总体数据创建了一个排名，如表 6-2 所示。

表 6-2 使用缓存获得的潜在收益（按页面排名）

排名值	页面	平均时间（秒）	每小时请求数	总耗时
1	/	0.55	700000	385000
2	/somePage	1.1	100000	110000
3	/otherPage	0.84	57000	47880

如果将主页面性能提升 5ms，则其最终效果要好于提升排第二位的页面的 10ms。如果我凭直觉，我很可能会一开始就在错误的地方进行优化和缓存，从而浪费大量时间。使用简单度量并对待处理页面进行排名，将主要精力放在最重要的页面上，结果会带来相当大的容量提升。

缓存失效的困难

“计算机科学中只有三件事最困难：缓存失效，命名事物，计数错误。”

——Phil Karlton

最后一个经验法则是缓存失效会很快变得困难起来。当你开始开发一个简单网站的时候，这看起来可能很容易。缓存失效只是当原始数据变化时，立即从缓存中删除该数据，避免使用到陈旧对象。你添加对象到缓存中，每当数据变化，就需要到缓存中删除陈旧数

据。很简单，不是吗？但不幸的是，大部分实际情况都比刚才说的要复杂得多。缓存失效很难，因为缓存对象通常是多个数据源计算的结果。反过来说，意思是无论任何原始数据有变化，你都需要将所有缓存对象进行失效处理。更复杂的是，内容的每个部分都可能多个呈现，这样所有片段都需要从缓存中删除。

为了更好地说明这个问题，我们考虑一个电子商务网站的例子。如果你很激进地使用对象缓存，你可能会对所有发送到数据存储的搜索结果进行缓存。你可以对分页产品列表，关键字搜索，类目页及产品页缓存查询结果。如果你想要保持缓存中数据的一致性，每当产品详情变化时，你就必须将所有包含那个产品的缓存对象进行失效处理。换句话说，你需要将所有查询结果进行失效处理，不仅包括产品页，也包括所有包含这个产品的其他列表和搜索结果。但不执行这些查询你怎么能确定这些查询结果是否包含该产品呢？怎样给所有类目列表构建这些缓存键，以及怎样在分页列表中确定正确的页号偏移量来让正确的对象失效呢？这就是问题所在——没有简单的方法来应对这个问题。

缓存失效最好的替代方案就是在缓存对象上设置一个较短的 TTL，这样数据就不会太陈旧。这样做通常是可行的，但有时也显得不够。在业务不允许数据不一致性的场景中，你也可以考虑缓存部分结果，然后到数据源获取缺失的关键信息。例如，如果业务需要你始终展示精确价格和可用库存，你仍然可以缓存大部分产品信息和复杂查询结果。仅有的额外工作就是在呈现结果前，从主数据存储中获取每个商品精确的库存和价格。尽管这样的混合方案并不完美，但它还是减少了数据存储需要处理的复杂查询的数量，只需要处理更简单的“WHERE product_id IN (...)”查询就可以了。

更高级的缓存失效技术超出了本书所讲的范围，但如果你感兴趣，我推荐阅读两个这几年发布的白皮书。第一个^[w61]解释了一个聪明的算法，是关于查询子空间失效的。第二个^[w62]描述了 Facebook 是如何通过在 MySQL 复制日志中添加缓存键来让部分缓存条目失效的。这个方法用于跨数据中心复制缓存失效命令，使数据存储更新后缓存及时失效。

由于失效性的本质，缓存失效问题通常很难复现和调试。尽管缓存失效算法很有趣，除非绝对必要我并不推荐去实现它们。我推荐尽可能避免笼统地进行缓存失效，而是使用基于 TTL 的过期策略。大部分情况下，加载重要数据时，较短时间的 TTL 或混合方案就已经足够满足业务需求了。

小结

缓存是最重要的伸缩性技术，它让你可以用相对较低的成本提升系统容量，还可以在开发的最后阶段添加到系统中，却无须大幅度地重构系统。如果你能在多个用户间重用缓存数据，或让请求不必到达服务器就能返回响应，那就说明你缓存用熟了。

大部分大型网站都重度使用了缓存，我强烈推荐你熟悉市面上的各种缓存技术，包括常见的 HTTP 缓存知识^[42]和 REST-ful Web 服务缓存。^[46]此外，也可以学习 Redis 技术，了解其在各方面的应用。^[50]

缓存在很多场景下都是一项古老的伸缩性技术。下面让我们将目光转移到一个近年来很流行的也更新鲜的概念上：异步处理。

7

异步处理

异步处理和消息技术引入了许多新概念和一些完全不同的软件处理思考方式。以前，我们需要告诉系统如何一步步执行，而现在，我们可以将任务切分并让系统决定最佳执行顺序。导致的结果就是执行更为动态化，结果变得难以预测。

如果使用得当，异步处理和消息技术可以在系统扩展和容错能力上提供强大支撑。然而熟练使用这些技术需要一个过程。本章会解释一些消息队列、事件驱动架构和异步处理技术方面的基础概念、利弊、陷阱，以及一些异步处理的实用技术。

本章结束后，你将能够更好地理解消息和异步技术如何工作。我也希望你因此对近年来非常流行的事件驱动架构更为着迷。

核心概念

在讨论异步处理前，我先解释同步处理及它们的区别。先看一些例子：

同步处理是软件执行的一种传统方式，在同步处理下，调用者发送请求，要求执行某个动作，然后等待执行响应，获得响应之后才继续自己的后续操作。

调用者通常依赖于该操作的执行结果，否则无法继续执行。调用者可以是一个函数，它调用其他函数、线程，甚至可能是进程。也可能是一个应用或系统向远程服务器发送请求。这些例子的关键是：调用者必须收到执行响应后才能继续执行自己的后续动作。

异步处理，简单说就是发送请求调用但不阻塞自己的执行。在异步模型中，调用者不需要等待它所调用的服务的响应，发送请求后马上继续后续处理而不会被阻塞。

同步处理的例子

我们以一个面向对象的邮件发送服务为例说明同步处理。假定我们有一个 `EmailService` 及一个方法 `sendEmail`，该方法接收参数 `EmailMessage` 并发送 E-mail。在代码清单 7-1 中，你能看到 `EmailMessage` 的结构和 `EmailService` 接口。这里我们忽略 `EmailService` 的实现，因为客户端只关心接口。

代码清单 7-1 简单的 `EmailService` 和 `EmailMessage` 接口

```
Interface EmailMessage {
    public function getSubject();
    public function getTextBody();
    public function getHtmlBody();
    public function getFromEmail();
    public function getToEmail();
    public function getReplyToEmail();
}

Interface EmailService {
    /**
     * Sends an email message
     *
     * @param EmailMessage $email
     * @throws Exception
     * @return void
     */
    public function sendEmail(EmailMessage $email);
}
```

当你想发送 E-mail 时，获取 `EmailService` 实例并调用 `sendEmail` 方法，`EmailService`

实现负责具体执行。例如，适配器 `SmtpEmailAdapter` 允许你通过邮件传输协议（SMTP）发送邮件。图 7-1 所示为调用时序。

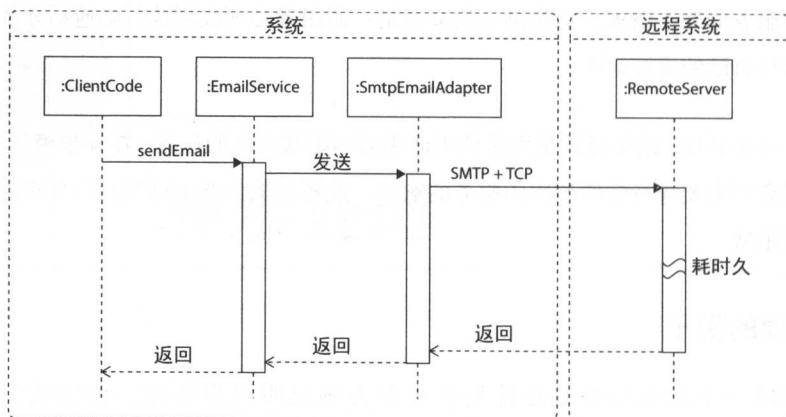


图 7-1 同步调用

这里需要强调的是你的代码必须等待邮件服务完成它的任务，也就是说，必须等待 IP 地址解析，网络连接建立，通过远程邮件服务器发送 E-mail，你还要等待消息编码和附件传输。这个过程通常需要花费好几秒，具体取决于 SMTP 服务器、网络速度、消息大小等因素。在这里，同步处理意味着代码必须和远程服务器保持同步协调，所有操作必须暂停一段时间以完成邮件发送。这种必须暂停执行等待响应的模式称为阻塞。

当你的代码需要等待外部操作完成时就会发生阻塞。当你从硬盘读取数据时会发生阻塞，因为操作系统需要时间去获取数据。当等待用户输入时也会发生阻塞，例如自动提款机在给你吐钱之前必须等你先插入信用卡。当你同步多进程/线程以避免竞争条件时，也有可能发生阻塞。

阻塞 I/O 意味着阻塞输入/输出，该术语用来描述对硬盘、网络连接或用户接口进行阻塞式读写。阻塞 I/O 大部分发生在与硬盘或网络连接交互时。例如，打开远程服务器的 TCP/IP 连接是一个阻塞操作（取决于你的编程模型），在这个场景下，你的线程会阻塞在打开一个网络连接的同步调用上。

同步处理使得构建一个响应及时的应用难度大大增加，因为没有办法保证阻塞操作到底需要多久才能执行完毕。每次执行阻塞操作，执行线程都会阻塞。阻塞线程在消耗资源，

却没有任何进展。在某些情况下，获取执行结果可能需要几毫秒，而有些时候则需要花费几秒，甚至得到一个失败的结果都有可能。

阻塞用户交互特别危险，因为用户很容易在等待过程中变得不耐烦。当 Web 应用页面失去响应 1~2 秒，用户就可能重新加载页面，单击“后退”按钮，甚至干脆关闭应用。企业应用的用户对关键商业处理的阻塞相对更有容忍度，因为他们必须完成工作，除了等待别无选择。另一方面，用户在上班路上浏览的时候是没耐心忍受等待的，如果你的应用强迫他们等待，你很有可能失去这些用户。

为了展示同步处理如何影响性能方面的感受，我们看看图 7-2，图中展示了所有阻塞操作是如何按次序一个接一个发生的。

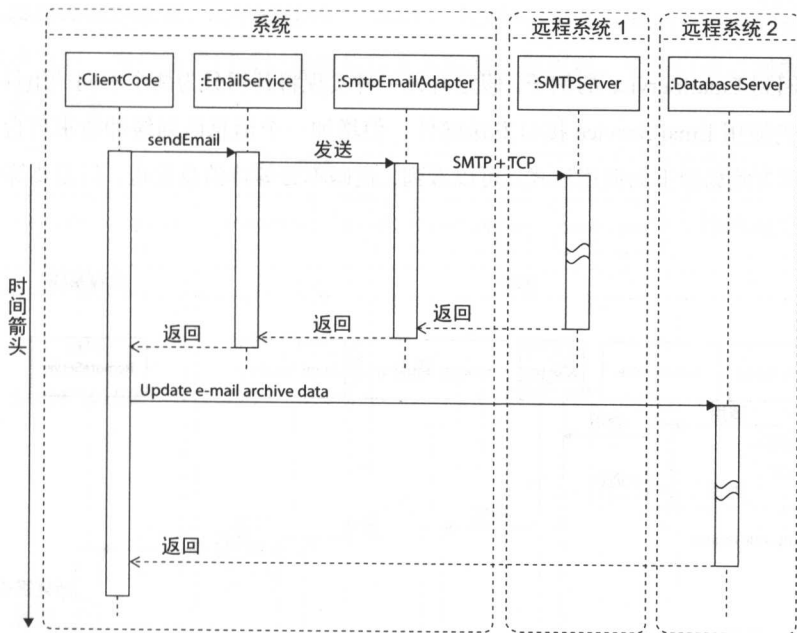


图 7-2 多个同步操作：累计执行时间

你执行的阻塞操作越多，你的系统就变得越慢，因为所有这些执行时间最后都会叠加起来。如果发邮件需要 100ms，更新数据库 20ms，那么你所有的执行时间至少是 120ms，在这个场景中，操作不是并行的。

现在我们解释了同步处理的情况，接下来我们再来看看同一场景异步处理的例子。

异步处理的例子

在一个纯粹的“发射后不用管”的模型中，客户端代码不知道请求会发生什么。客户端可以继续完成自己的工作，甚至不知道请求是否被处理。异步处理并不总是“发射后不用管”模型，异步调用结果也可以采用回调机制被调用者消费。

回调是一种异步处理技术，调用者等待执行结果时不会阻塞，而是提供一种操作完成后被通知的机制。回调是一个异步操作完成后被调用的函数、对象或端点。例如，如果一个异步操作失败，回调允许调用者处理错误信息。回调在用户接口环境下更为适用，因为它们允许慢任务在后台执行，和用户交互并行进行。

我们回到 EmailService 的例子，假定有另一种实现将其切分为两个不同的组件。我们在客户端仍使用 EmailService 接口发送邮件，但增加一个消息队列缓冲请求后台处理。图 7-3 所示为此场景下的调用形式。可以看到，代码不必等待消息发送，只需要等待消息被插入消息队列中。

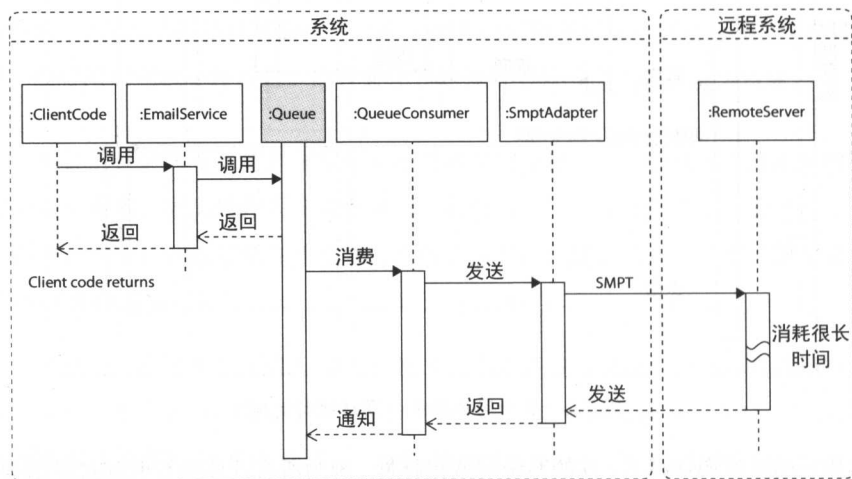


图 7-3 E-mail 消息异步处理

你的代码并不知道 E-mail 是否被成功发送，因为代码执行结束时，邮件还没有被发送。它只是添加到队列中等待被 SMTP 服务器发送，这是一个“发射后不用管”形式的

异步调用的例子。

另一个需要注意的是我们可以有独立的线程/进程。客户端代码可以运行在一个独立的执行进程中，并在任意时间点将消息添加到队列中。另一方面，消息队列的消费者，可以在另外的进程中以不同的速率发送 E-mail。消息消费者甚至可以被关闭或者崩溃，而只要消息被放入队列中，客户端代码就无须知晓消息是否被处理。

如果想处理 EmailService 发送邮件的结果，我们可以提供一个 Web Service 端点，或其他通知方式（某种回调形式）。这种方式下，每当 SMTP 请求失败，或监测到邮件退回，我们都能被通知到，然后实现回调功能处理这些通知。例如，可以在数据库中标记邮件发送记录是成功或退回，并且基于这些状态，通知用户发送失败。显然，根据业务需求，回调既能够处理成功通知，也能够处理失败通知。如图 7-4 所示，我们可以看到引入回调功能后的序列图。客户端代码可以继续执行而不阻塞，但同时，它可以通过回调处理各种 E-mail 发送结果。

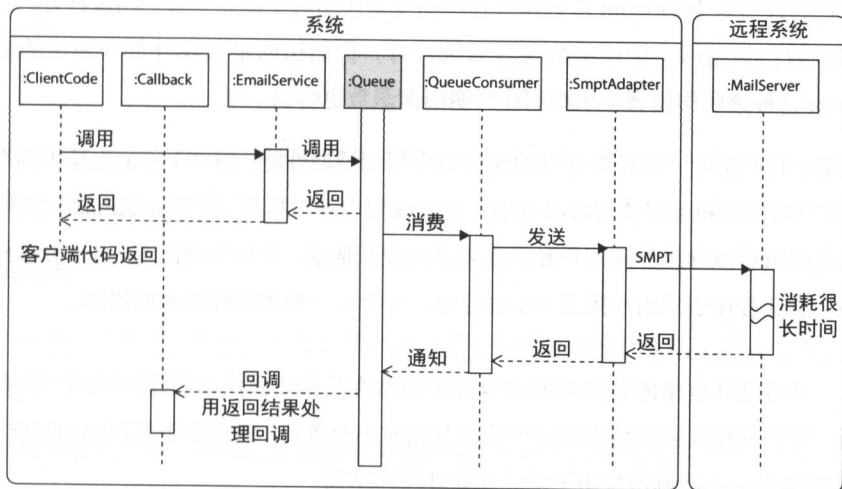


图 7-4 回调式异步调用

图 7-4 已经对实际实现做了简化，但我们可以看到它依然比图 7-1 所示的版本复杂。这是因为我们将调用序列有效地解耦为几个独立应用，而不再是单个应用。我们可以让客户端代码、回调、队列、队列消费者在不同线程执行，而不再是所有步骤只在单一线程中完成。它们也可以在不同服务器的不同进程中执行。

为了让异步处理更简单,需要提供简单框架和分层抽象,将异步调用和回调的路由分发隐藏起来,这一点非常重要。AJAX 是一个很好的例子,它让异步处理使用非常简单。如果一个 E-mail 消息由浏览器中运行的 JavaScript 发送,我们可以提供一个回调函数来处理结果。代码清单 7-2 所示为一个发送 E-mail 并带有传参回调函数的例子。

代码清单 7-2 带有回调函数的 sendEmail 函数调用,回调函数声明放在主函数里

```
// messageRow 变量被提前声明,并绑定到屏幕的 UI 元素上
emailService.sendEmail(message, function(error){
    if(error){
        // 访问 messageRow 修改 UI
        messageRow.markAsFailed(error);
    }else{
        // 访问 messageRow 修改 UI
        messageRow.markAsDelivered();
    } });
```

此处的技巧是 JavaScript 匿名函数在声明变量作用域中捕获变量。用这种方式,即便在外部函数已经返回,回调函数在后面执行时,仍能访问定义在外部作用域的变量。JavaScript 这种透明作用域继承机制让声明回调函数更容易。

最后,我们思考一下异步处理如何影响应用的预期性能。图 7-5 所示为异步调用怎样执行。客户端代码和远程调用可以分别在单独线程中并行执行。一旦 sendEmail 方法返回,客户端代码就可以继续处理用户和页面元素的交互请求。并行处理由 JavaScript 事件循环来模拟,它让程序表现出非阻塞 I/O 的行为,并给人一种多线程处理的错觉。

非阻塞 I/O 指的是输入/输出操作不阻塞客户端代码执行。当使用非阻塞 I/O 库,代码从磁盘或网络套接字中读取数据时不会等待。执行非阻塞调用的时候,需要提供一个回调函数用于响应并处理操作结果。

在这个场景下,我们制造了一个 E-mail 被立即发送的假象。一旦用户单击按钮,sendEmail 函数被调用,异步处理就开始。用户被立即告知 E-mail 被接受,可以继续其他工作了。即便发送 E-mail 需要 100ms,更新数据库需要 20ms,用户也并不需要等待这些步骤全部执行完。如果必要,回调代码执行时,它会通知用户消息是否被发送成功。

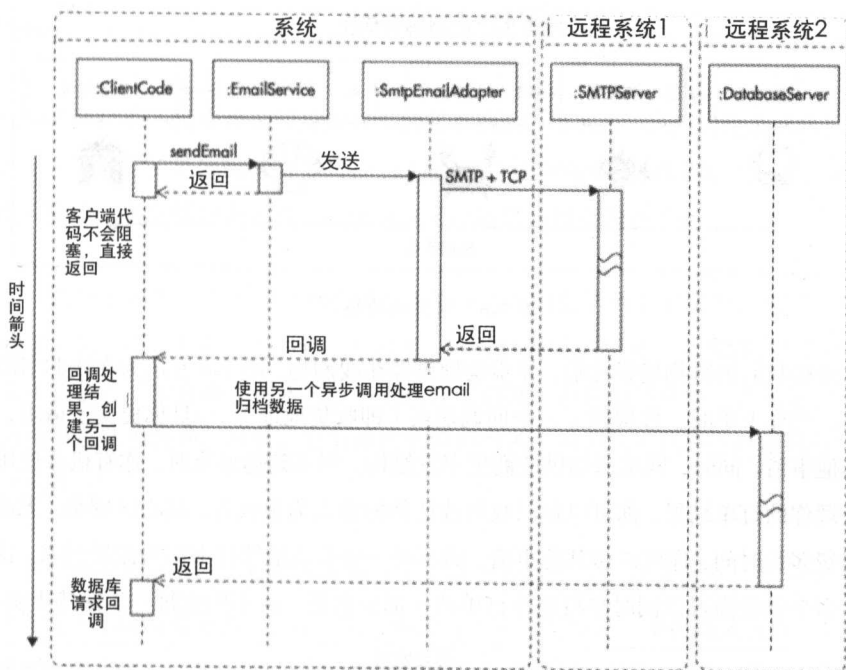


图 7-5 多异步操作：执行时间对用户隐藏

我们已经讨论了同步和异步处理模型的核心概念，接下来，我们用一个快速类比来进一步简化这个复杂的主题。

购物类比

为了进一步简化，你可以这样想象同步处理，就像你在水产市场上买鱼一样。你走进一家鱼店，要买一条鱼，然后等待。

鱼贩为你包好鱼，并问你是否需要其他东西。你要么买更多海鲜，要么去下一个采购地点。不管你要买多少东西，你一次只能买一件商品。在你去下一个采购地点买螃蟹前必须先买到鱼。图 7-6 所示就是这样一个场景。你可能会问为什么我们用卖鱼的市场做例子？只是为了更有趣，更容易记忆。

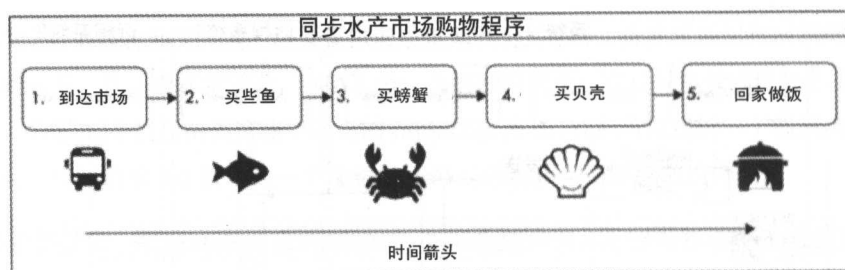


图 7-6 同步购物场景

继续聊我们的购物场景比喻，异步购物更像在线购物。图 7-7 所示为在线购书时的事件序列。当你下单时，你提供了一个回调端点（即收货地址）。一旦你提交了请求，你可以忙其他事情。同时，网站通知供应商把书发给你。当书到达你家时，你有机会使用回调功能处理你的订单结果。你可以自己收货或让你的家人为你收货。核心区别是，无论快递需要花费多长时间，你可以做其他事情，而不必一动不动地等待书籍被派送到家。这也意味着，多个供应商可以同时平行地为订单的一部分供货，而不用在每一步保持步调一致。

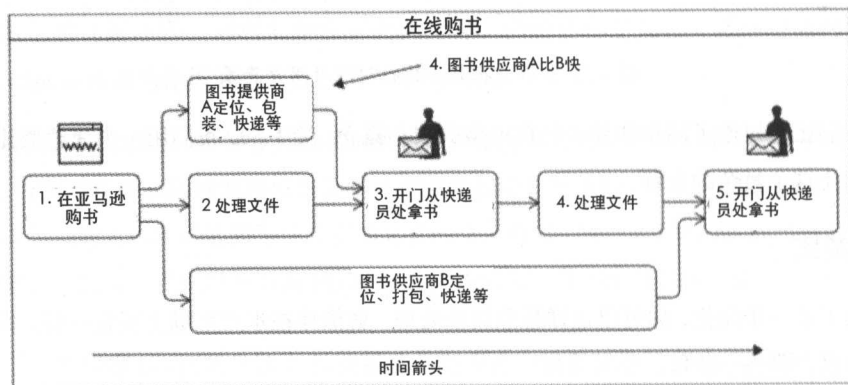


图 7-7 异步购物场景

除此之外，如果你是为朋友订购书籍，那么你完全不用响应后续流程，你的订单就是一劳永逸的请求。

从伸缩性角度讲，这两种方式的主要区别是：更多的代理（进程、线程或独立系统）可以在同一时间点并行工作。这也意味着，你可以在独立的 CPU 或独立服务器上执行每个代理程序。

消息队列

我们已经讨论了同步处理和异步处理的基本概念，接下来，我们讨论消息队列。消息队列是一个实现异步处理的有力工具，可以在基于同步模型实现的应用中采用。即便你的应用或编程语言不支持异步处理，你也可以采用消息队列实现异步处理。

消息队列是一个缓冲并分发请求的组件。在消息队列上下文中，假定消息是单向的，即“发射后不用管”的请求。你可以假定消息是一串 XML 或 JSON 文本，包含了所有需要用来执行请求操作的数据。消息由消息生产者创建，然后由消息队列缓冲。最后，它们被推送给消费者，由消费者代表生产者执行异步动作。

一个可伸缩系统中的消息生产者和消费者通常作为独立进程或线程运行。生产者和消费者通常位于不同的服务器上，可以用不同的技术来实现以进一步提高灵活性。生产者和消费者可以彼此独立工作，两者只会通过消息格式和消息队列产生耦合。图 7-8 所示为生产者创建消息，并发送到消息队列。与生产者独立，消息队列以一定次序排列消息并发送给消费者，然后消费者从队列中获取并消费消息。

这是消息队列的一个非常抽象的描述。这里我们不关心消息队列的实现、生产者怎样生产消息，以及消费者怎样消费消息。在这个抽象级别，我们只关心消息的总体流动，以及生产者、消费者如何通过消息队列解耦。

分离为生产者和消费者解耦提供了一种非阻塞通信的好办法。生产者不用等待消费者，生产者线程不用阻塞直到消费者准备好处理消息。相反，生产者向队列提交作业请求，这样能更快地响应请求，因为不需要立即处理。

这样解耦的另一个好处是生产者和消费者可以独立伸缩，这意味着，可以在任何时间增加更多的生产者而不会让系统过载。如果消息不能被及时消费，可以对队列做线性伸缩。我们还可以独立地伸缩消费者，将他们部署到不同的机器上，消费者数量也能够独立于生产者增长。图 7-8 所示为三种不同的职责：生产者、消息队列和消费者。接下来，我们更详细地看看每种职责。

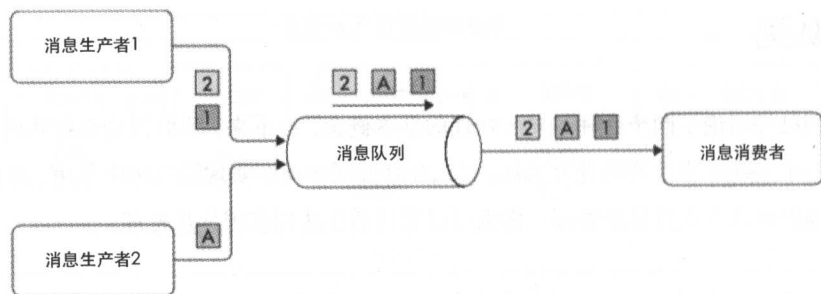


图 7-8 消息生产者、队列和消息消费者

消息生产者

消息生产者是客户端的一部分代码,用来初始化异步处理。在基于消息队列的处理中,生产者职责很少,它们要做的就是创建一条合法消息,并发送到消息队列。由应用开发者决定生产者在哪里执行,什么时候发布消息。

生产消息一般称作发布或发布消息。消息生产者和消息发布者是同义词,可以互换使用。

应用通常有多个生产者,在不同的代码中发布相同类型的消息。所有消息被排队和异步处理。

回到之前的 EmailService 例子,如果邮件服务使用消息队列实现,则生产者是邮件发送客户端代码的实例。代码中的生产者会处理账户创建,购买确认和密码重置。当你想发送 E-mail 时,你可以生产一条消息并添加到队列中。生产者可以用任何技术实现,只要它能定位消息队列并添加一条消息到队列。代码清单 7-3 所示为一个示例消息的样子。消息格式是生产者和消费者之间的约定,所以严格定义和验证消息格式非常重要。

代码清单 7-3 定制消息格式,生产者和消费者之间的约定

```

<?xml version="1.0"?>
<emails>
  <message>
    <type>NEW-ACCOUNT</type>
    <from>some@guy.com</from>
    <to>your@client.org</to>
  
```

```
<subject>Welcome to Our.Service.Com</subject>
<textBody>
    Contents of the message.
</textBody>
<htmlBody>
    &lt;h1&gt;Contents of the html.&lt;/h1&gt;
</htmlBody>
</message>
</emails>
```

使用平台独立的格式比如 XML 或 JSON 允许生产者和消费者使用不同的技术实现，并彼此独立工作。当用户向新账户订阅邮件时，你可以用 PHP 代码创建 E-mail，也可以用 Java 编写后台系统，每次购物时发送邮件。这两种生产者可以创建 XML 格式消息并发送到消息队列中。生产者不用等 E-mail 被实际发送，它们可以简单地假定 E-mail 总会在某个时间点被发送出去。

提示

不用知道消费者如何实现，采用什么技术，甚至它们是否可用，等等，这些都是非常棒的解耦合设计的特征。

正如我们之前所讲，消息生产者很自由且职责很少。

接下来，我们看看消息队列。

消息代理

基于消息队列的异步处理技术的核心组件就是队列本身，它是消息发送的目的地和发给消费者的缓冲。消息队列可以用不同的方式实现。最简单的方式是用共享文件夹实现，通过从文件夹读写文件实现消息队列。消息队列也可以是 SQL 数据库支撑的组件（许多公司自己开发的消息队列），也可以是支持接收、路由、持久化和发送消息的专用消息代理。消息队列还可以是同一应用进程中的一个单独线程。

由于消息队列是一个有诸如权限控制、路由、失败恢复等多重职责的独立组件，所以通常被实现为一个独立应用。在这种场景下，它通常被称为消息代理或面向消息的中间件。

消息代理是一个专门的应用，用于快速灵活地处理消息排队、路由和分发。消息代理是消息队列的一种更复杂的实现方式，通常提供一些特定的功能。由于消息代理的一个核心职责是消息排队，所以它们为高并发、高吞吐环境做了专门优化。根据实现的技术不同，消息代理也被称为面向消息的中间件（MOM）或企业服务总线（ESB）。虽然 MOM 和 ESB 需要承担非常多的职责，但它们的目的都非常简单。

消息代理比生产者职责更多，它是解耦生产者和消费者的关键元素。消息队列的主要职责是保证生产者任何时间都可以接收消息。同时负责缓冲消息，并让消费者消费消息。与 Web 应用或数据库引擎类似，消息代理也是应用。消息代理通常不需要任何定制代码，只需要配置参数，比通常的数据库简单，这让它能达到更高的吞吐能力和伸缩能力。

由于消息代理是独立组件，在伸缩时它们有自己的需求和限制。不幸的是，引入消息代理会增加系统架构的复杂度，需要我们恰当使用和伸缩。接下来，我们会讨论使用消息代理的优缺点，首先还是看看消息消费者。

消息消费者

现在，我们讨论最后一个组件：消息消费者。消息消费者的主要职责是从消息队列中接收并处理消息。消息消费者由应用开发者实现，是实际处理异步请求的组件。

回到 EmailService 服务的例子，消费者代码负责从队列中获取消息，并通过 SMTP 协议发送到远程邮件服务器。消息消费者类似生产者，可以用各种技术实现，独立修改及运行在不同服务器上。

为了实现高层解耦，消费者不应该知道生产者的情况。它们只依赖消息队列中的合法消息。如果我们遵循这条规则，可以将消费者放到更低的服务层，这样就变成了单向依赖。生产者依赖于“某些消息消费者”的动作，但反过来，消费者并不依赖生产者。

消息消费者通常部署在不同的服务器上，并通过添加额外硬件独立于生产者进行伸缩。实现消费者最主要的两种方式是周期模式和守护模式。

周期模式下，消费者定期连接到消息队列检查队列状态。如果有消息就消费；如果队列为空，就消费一定数量的消息后停止。这个模型在脚本语言中更常见，例如 PHP、Ruby 或 Perl，这个场景下没有持久运行的应用容器。周期模式也称为拉模式，因为消费者从队列中拉消息。也可以用在队列中消息较少或网络不稳定的场景。例如，移动应用的连接可以假定在任意时间断掉，导致它可能一次又一次地从队列里拉消息。

守护模式下，消费者以无限循环方式运行，通常和消息代理有一个持久连接。不同于周期地检查队列状态，它只是简单地在 Socket 读操作上阻塞。这意味着消费者空闲时等待，直到消息被消息代理推送过来。这种模型在拥有持久化应用容器的语言中更常见，例如 Java、C#、Node.js。因为消息按消费者处理速度被消息代理推送给消费者，因此也被称为推模型。

这两种方式相互并无明显优劣，它们只是同一个问题的两种不同解决方式，即如何从队列读消息并处理消息。

除了不同的执行模型，消费者还可以使用不同的订阅方法。消息代理通常允许消费者指定他们感兴趣的消息。可以直接从一个命名队列中读取消息或使用高级路由方法。不同路由方法的可用性取决于你要用哪个消息代理，通常支持以下路由方法：直接工作队列、发布订阅、定制路由规则。^[12, 24]

接下来，我们快速浏览不同的路由方法。

直接工作队列模型

在这种发送模型中，消费者和生产者只需要知道队列的名称。生产者发送的每条消息被发送到单个队列。队列由名称定位，多个生产者可以在任何时间点向它发送消息。在队列的另一端，可以有一个或多个消费者竞争消费消息。每条到达队列的消息只被路由到一个消费者。这种模式下，每个消费者只能看到消息的一个子集。图 7-9 所示为直接工作队列的结构。

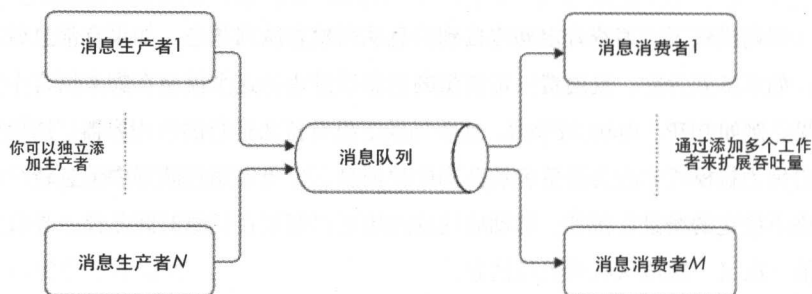


图 7-9 直接工作队列

这种路由模型非常适合在多台工作机器上分配很耗时的任务。消费者无状态并且形式统一，这样失效节点替换就会像添加新节点一样容易。伸缩也变得不那么复杂，因为我们要做的就是添加更多的工作机器来提升整体消费吞吐量。请注意，消费者可以独立于生产者进行伸缩。

适用这个模型的例子包括发送邮件、处理视频、调整图片大小或向第三方 Web 服务上传内容。

发布订阅模型

在发布订阅模型中，消息可能被发送到不止一个消费者。生产者发送消息到一个主题，而不是队列。消息到达一个主题，然后被克隆给每个订阅了该主题的消费者。如果在发布的时间点没有消费者，消息可能被全部丢弃（取决于消息代理的配置）。

使用发布订阅模型的消费者必须连接到消息代理，并声明它对哪个主题感兴趣。当消息发布到主题，就被复制给每个订阅它的消费者，每个消费者接收一份消息复制到自己的私有队列。消费者可以独立于其他消费者使用消息，因为它的私有队列复制了指定主题的全部消息。

图 7-10 所示为发布到主题的消息如何被复制到属于每个消费者的不同队列中。

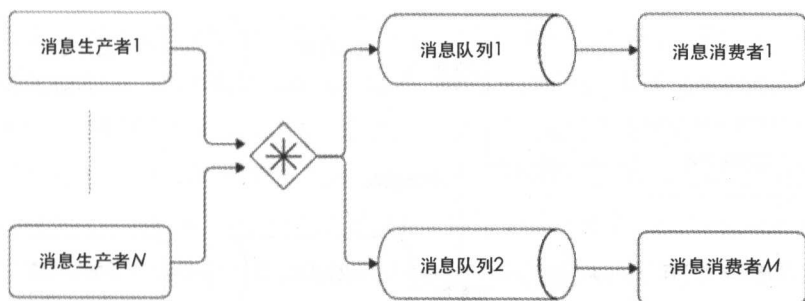


图 7-10 发布订阅队列模型

这个路由模型的例子是为每次购物发布消息。电子商务应用可以在每次购物被确认后，发送消息到消息队列的某个主题。为了处理购物消息，你可以创建不同的消费者执行不同的动作。你可以创建一个消费者通知货物供应商，一个消费者处理风控规则，还有一个消费者分配奖励积分。你也可以在未来添加更多功能而不用修改现有的发布者或消费者。如果你想添加一个消费者发送一封 PDF 格式的购物确认 E-mail，可以简单地部署一个新消费者订阅同一个主题。

发布订阅模型是消息技术中一种比较灵活的模式。它也是一个通用设计模式——观察者模式的变体^[1,7]，用于组件解耦和践行开闭原则（第二章有叙述）。为了让这种模式更灵活更易伸缩，很多消息代理允许消费者相互竞争，在这个场景下，多个消费者订阅同一个队列，消息在它们之间分配，而不是一个消费者处理所有消息。

定制路由规则

有些消息代理也支持各种形式的路由定制，消费者可以选择更灵活的方式决定消息如何路由到自己的队列中。例如，在 RabbitMQ 中，可以绑定一个概念来创建灵活的路由规则（基于文本模式匹配）。^[12]在 ActiveMQ 中，可以使用 Camel 扩展来创建更高级的路由规则。^[25]

关于基于模式匹配定制路由，一个更好的例子是日志和报警。你可以在票务系统创建一个“日志队列”接收日志消息，一个“报警队列”接收严重错误和所有级别为 1 的消息。然后，你可以创建一个“文件日志器”简单地将“日志队列”的所有消息写到一个文件。你还可以创建一个“报警生成器”接收所有“报警队列”的消息生成操作通知。图 7-11 所示为一个这样的配置。

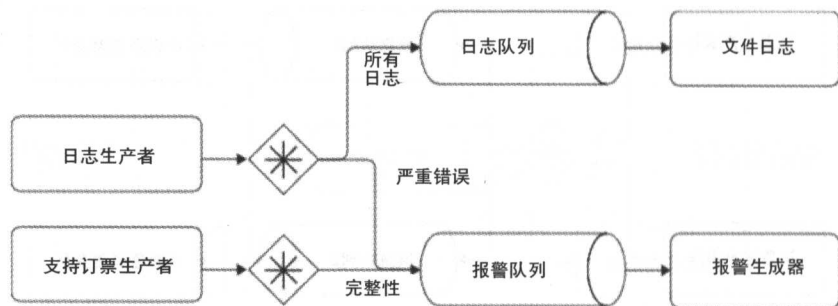


图 7-11 定制路由配置

定制路由背后的思想是增强消费者订阅消息的灵活性。消息代理灵活性增强，你的系统就可以采用配置实现新需求，而不用修改已有的生产者和消费者代码。

以上就是一般的路由方法，但我鼓励你阅读消息代理的官方文档，或者是一些消息技术方面的书籍。^[12,24,25] 现在我们已经了解了异步处理和消息最重要的概念，接下来我们看看不同的消息协议，然后在基础设施层面看看消息代理。

消息协议

消息协议定义了客户端程序如何连接到消息代理，以及消息如何传输。协议可以是二进制或文本，它们可以指定最基本的功能，也可以详细描述上百个特性。你可能对在生产者和消费者之间传输消息的协议很熟悉。作为一个应用开发者，你可能用不着开发消息协议的任何实现，但最好还是理解每种协议的特性，这样你可以选择最适合你的项目的协议。这里，我们看看开源世界最常用的几种协议：AMQP、STOMP 及 JMS。

AMQP（高级消息队列协议）是一个明确定义消息发布、消费和传输的协议，它是行业标准。它比 STOMP 更高级，定位于企业集成和互操作。自从它作为标准协议被 OASIS（高级结构化信息标准组织）接受，^[W54] 集成不同消息供应商、消费者和发布者就变得更加容易了。AMQP 在协议规范中包含了很多特性，例如可靠消息、传递保证、事务及其他高级特性，并确保所有支持的库和服务都以相同的方式实现。许多现代编程语言都有稳定的 AMQP 客户端，其中 RabbitMQ 和 ActiveMQ 都支持 AMQP 作为通信协议。考虑到这些好处，如果可能，我推荐你尽量使用 AMQP 作为消息协议。

STOMP（流式面向文本的消息协议）是一个真正极简化的协议。事实上，简单是其最主要的优势之一。STOMP 是一个无状态的，类似 HTTP 的，基于文本的协议。它支持较少的操作，因此开发和调试更简单。这也意味着协议层不需要更多的性能开销。STOMP 令人不满意的地方是高级特性需要通过定制头信息来扩展实现，这导致互操作受限，因为特定行为没有统一的标准。消息预取数量就是互操作受限的很好的例子。预取数量是指消费者可以提前声明它想从服务器获取的消息数量而无须每次通知它们。预取是提高吞吐量的一个好办法，因为消息是批量获取而不是每次只取一条。尽管 RabbitMQ 和 ActiveMQ 支持这个特性，但它们使用了不同的 STOMP 定制头来实现。如果与 ActiveMQ 交互，你需要用“`activemq.prefetchSize`”头参数指定；如果与 RabbitMQ 交互，需要设置“`prefetch-count`”。显然，这使你无法创建一个通用的 STOMP 客户端程序库来实现预取特性，因为你的程序库需要知道如何与消息代理的每个部分协调。更糟糕的是，你的代码需要知道它是跟 RabbitMQ 还是 ActiveMQ 在交互。即便这个例子非常简单，它依然说明了标准的重要性，以及与非标准协议集成的困难性。

最后一个协议，JMS（Java 消息服务）是一个 Java 消息标准，在基于 Java 的应用服务器和程序库中被广泛采用。虽然 JMS 提供了一组很好的特性集，不过不幸的是，它是一个纯 Java 标准，因此与非基于 JVM 的技术进行集成很受限制。如果你的开发基于纯 Java 或基于 JVM 的语言比如 Groovy 或 Scala，那么 JMS 对你来说是个不错的协议。如果你要与不同的平台集成，那么你最好用 AMQP 或 STOMP，因为它们为所有流行的语言提供了实现。

从伸缩性角度看，传输消息的协议不是关键，所以你应该基于功能集、工具或程序库的可用性选择你的编程语言。

消息基础设施

到目前为止，我们已经讨论了消息队列、代理、生产者和消费者。我们也描述了几个流行的消息协议。接下来，我们再看看消息组件怎样影响我们的系统基础设施。

我们首先看看第一章中的基础设施。图 7-12 所示为第 1 章中的基础设施，不过为了更清晰，消息代理被加亮显示。消息队列系统通常可以同时从前端系统和后端系统访问。通常是在前端系统生产消息，在后端系统消费消息，但不一定都是这样。有些应用可能在

前端消费消息。例如，一个在线聊天应用可以在消息到达邮箱时消费消息通知用户。最终，你怎样使用消息取决于你的需求和用例——这只是工具集中的工具而已。

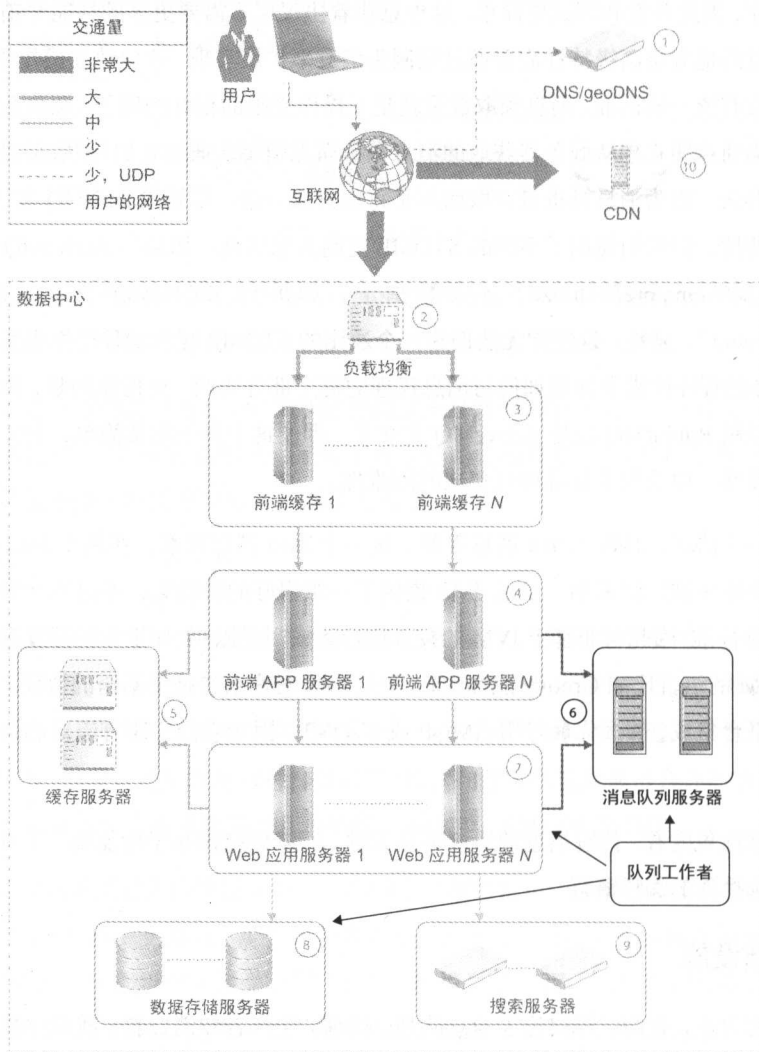


图 7-12 系统基础设施中的消息代理和队列工作者

在图 7-12 中，消息消费者服务器标记为“队列工作者”。经常可以看到整个服务器集群只专门负责消息处理。这些机器称为队列工作者，因为它们的唯一职责是执行消息处理。

提示

如果你将服务器放于云端，像 Amazon EC2 或其他可视化提供商那样，你可以根据任务处理瓶颈（内存、I/O 或 CPU）很容易地为队列工作者集群选择不同的服务器实例。

最好将队列工作者分隔为几组不同的服务器，这样它们的伸缩性就不依赖于其他组件的伸缩性。队列工作者越独立，就越少受系统的其他部分的影响。重要的一点是，队列工作者机器必须无状态，就像 Web 应用服务器和 Web 服务主机。工作者从队列或外部存储中获取所有数据进行处理，这样机器失效和伸缩扩容就都不是问题。

提示

为了保证队列工作者无状态，你可以使用其他服务来保存和获取状态。例如，如果工作者的任务是转码视频，消息生产者首先会上传视频二进制文件到分布式存储（例如 S3、共享 FTP、SAN 或 NAS），然后将视频文件的位置发布到消息队列中，这样队列工作者机器就不用保持本地状态来处理消息。

通过让队列工作者无状态及机器分组，你可以简单地添加机器实现水平伸缩。机器失效对你没有影响，因为新工作者可以很容易地添加到集群代替失效机器。

通常，消息代理提供一些内建的水平伸缩功能，但每种消息代理产品可能都有自己的陷阱和伸缩限制。单个队列的总吞吐量是有限制的，因为通过队列的消息要被发送给所有连接的订阅者，但是只要可以使用应用级分区在多个队列间分发消息，你就可以通过添加更多的消息代理服务器进行水平伸缩。

如果你需要处理每秒数万级别消息的吞吐量，RabbitMQ 或 ActiveMQ 应该能胜任。但是如果你计划处理每秒 10 万级别的消息，你就需要自己定制分区机制将负载分散到多个消息代理实例。

令人诧异的是，即便使用基于云的消息平台比如微软 Azure Queues，也可能遇到伸缩限制。例如，微软 Azure Queues 每个队列每秒消息限制数为 2000，这已经很多了。^[11]另一个称为服务总线队列的 Azure 产品单个队列有 100 的并发连接限制。可能你的需求与此

不相干，但除非你做了必要的处理，否则不能简单地假定消息队列可以无限伸缩。在选择一个消息解决方案前，通常需要考虑价格、基础设施需求、伸缩保证等因素。

提示

你可以认为消息代理是一个非常笨的 SQL 数据库引擎——不能更新数据，只能执行在表尾添加数据，在表头弹出数据。消息代理既可以独立于应用也可以嵌入在应用中。消息代理只是一个向队列添加消息，向消费者发送消息的抽象概念。

在选择消息代理软件及担心是否需要伸缩之前，先对应用做以下评估：

- 每秒发送的消息数
- 平均每条消息的大小
- 每秒消费的消息数（这可能比发布率更高，因为多个消费者可能订阅同一消息的不同副本）
- 并发发布者数量
- 并发消费者数量
- 是否考虑消息持久化（消息代理崩溃情况下，消息不丢失）
- 是否需要消息通知（消费者崩溃情况下，消息不丢失）

经过这些评估，你就有了伸缩性需求的依据来选择供应商或开源软件。接下来，我们看一些具体的消息代理并讨论它们在伸缩性方面的影响。在开始前，我们先回顾给系统添加消息机制的好处和动力。

消息队列的好处

目前为止，我们已经看到了异步处理和消息队列的核心概念和术语，你可能认为它们并不是轻而易举就能获得的。你可能需要学习、部署、优化和伸缩你的消息队列。在技术栈中添加这些组件往往会增加系统整体的复杂性。既然这么麻烦，为什么还要使用消息队列？以下是引入消息队列的几点好处：

- 异步处理
- 易伸缩

- 使峰值变平缓
- 隔离失效机器及自我修复
- 解耦

除了这些好处，消息队列也是一个非常特别的技术。一旦你熟悉了消息队列，你就会找到许多非常适合消息队列的应用场景，让事情变得更简单、更快速。

实现异步处理

使用消息队列的一个显而易见的好处是可以推迟耗时操作的处理，而不必阻塞客户端。消息代理成为异步处理世界的入口，任何执行缓慢或不可预测的任务都可以考虑用消息队列进行异步处理。唯一要操心的就是你需要找到一种方法，不用取得慢操作结果就能使客户端代码继续执行。

适合消息队列的应用场景包括：

- 与远程服务器交互

如果你的应用在远程服务器上执行操作，通过队列来推迟这些操作就很有用。例如，如果你有一个电子商务平台，允许用户创建营销活动来宣传他们的产品，在这个场景下，你可以让用户发布要推广的商品，并把它们添加到队列，这样用户就不用等待远程服务调用执行完毕。在消息消费者后台，系统可以联系多个广告供应商，例如 Google AdWords 开展营销活动。

- 低成本处理关键路径

每个应用都有一些关键路径或特性，其拥有最高优先级，必须一直执行。在一个电子商务系统中，可能是下订单、搜索商品和处理支付。关键路径的一个主要要求是：即便其他部分崩溃也必须 7×24 运行。简单说来，如果不能下单或支付，这个电子商务系统有什么用？在这些约束条件下，如果在结账功能上集成一个新的推荐引擎，就可能引入一个新的失效点，它可能拖慢结账处理本身。与同步发送订单到推荐系统获取推荐信息不同，你可以将推荐请求放入消息队列，并在一个独立组件中异步处理。

- 资源密集型工作

许多 CPU 或 I/O 密集型处理，例如视频转码、图片大小调整、构建 PDF 或生成报

表，都非常适合使用基于队列的工作流，而不是在用户交互时同步处理。

- 独立处理高优先级或低优先级作业

例如，可以将队列分隔为高级消息（紧急任务）和低价值消息（非紧急任务）。

然后，可以将资源专注于高级作业，保护它们不受低价值任务的影响。

消息队列能让你的应用以异步方式工作，但仅对一开始以非异步方式构建的应用才有价值。如果你在异步环境开发，例如使用 Node.js——它的内核本来就是异步的——这种场景下引入消息代理并不会增加更多的收益。消息代理并不能让已经是异步的系统更异步。消息代理的好处是可以将异步处理引入其他平台，例如天生是同步模式的平台（C、Java、PHP、Ruby）。

更好的伸缩性

如图 7-13 所示，由于采用延迟处理模式，使用消息代理的应用非常容易伸缩。既然生产者消息是“发射后不用管”的请求，耗时的任务就可以向多台服务器并行地发布请求，然后在多台后台服务器上并行地处理消息，即多台物理机上运行消息队列消费者，当负载上升时，很容易就能添加进更多的机器。

一个并行后台处理的场景是调整图片和视频大小。前端应用通过网络上传文件，存储于 NAS①，待处理的文件通过消息发布到队列②，消息被缓冲在消息队列，在随后的阶段被工作者提取处理③。每个工作者从队列中取出一条消息，进行图片处理（需要花费一些时间）④。工作者也可以向指定的队列发送一条消息表明工作已经完成。在这个配置中，当需求变化时，可以很容易地添加和删除后台服务器。

正如你所见，通过添加更多的消息消费者，可以增加整体吞吐量。无论每台工作者节点自身有怎样的限制，我们都可以通过添加更多的工作者服务器处理更多的消息。更灵活的地方在于：添加消费节点不需要变更生产者配置。消费者仅需要连接到消息代理，开始处理消息；生产者并不需要知道有多少消费者或它们的位置。

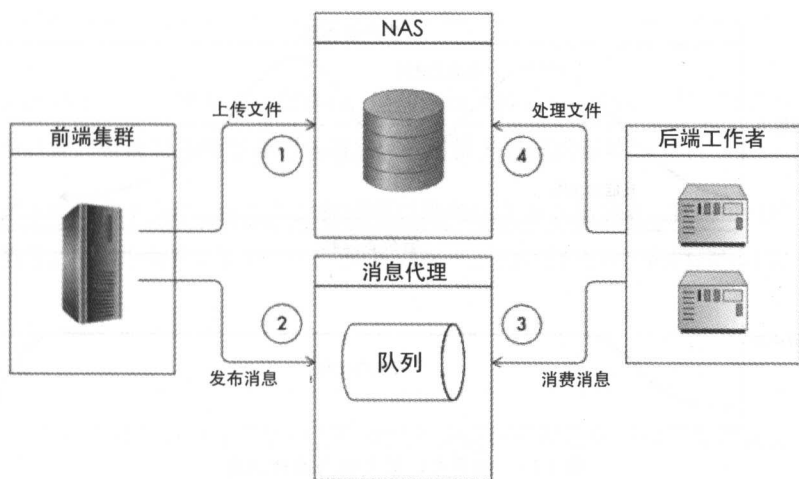


图 7-13 通过添加更多并行工作者进行扩容

即便你使用 Node.js 或 Erlang，它们本身就是异步的，但你仍然能通过使用消息队列获益，可以在多台服务器中有效分担工作负荷。

平衡流量峰值

使用消息队列的另一个好处是：它可以透明地平衡流量峰值。使用消息队列，即便流量持续增长，仍然可以持续以高频率接收请求。在这种情况下，虽然发布消息的速度比消费速度快，但是可以持续地将消息纳入队列，因此短时间内发布者不会受消费者处理能力的影响。

如果你的前端应用生产消息，后台集群消费消息，那么前端流量越多，发布到队列中的消息就越多。既然前端代码不必等待慢操作的完成，消费者是否能立即处理消息，都不会影响前端用户。因此即使生产消息的速度快于消费的速度，消息仍然能很快放入队列。流量峰值唯一的影响就是消息从发出到处理需要更长的时间，因为消息在队列中停留时间更长。图 7-14 所示为有待处理消息时，消费者满负荷工作时的状态。即便前端应用生产消息超过了消费者的能力限制，消息仍能入队并被逐渐处理。当流量峰值过后，消费者最终能跟上消息产生的速度，并处理队列剩余消息。

平衡流量峰值的特性提高了可用性。即使系统无法立即处理所有请求也不会崩溃。峰值过后，系统很快自动恢复到平常状态。

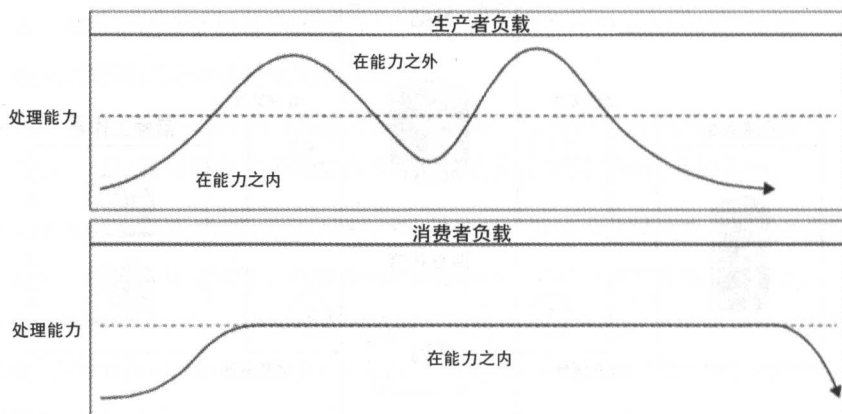


图 7-14 消费者以最大能力处理消息

失败隔离和自我修复

正如之前讨论的，消息队列允许我们从关键路径上删除一些功能，将生产者和消费者隔离，提高系统的健壮性和容错性。因为发布者不直接依赖消费者，所以消息系统可以将消费者系统错误与生产者系统组件隔离。生产者不会受消费者一端失败的影响。相应地，即便生产者遇到技术问题，消费者也能继续自己的工作。只要消息在队列中，消费者就不会受生产者失败的影响。

生产者可用性不受消费者影响这一点允许我们在任意时刻停止消息处理。这意味着任意时刻，我们都可以对后端服务器执行维护和发布操作。我们可以重启、添加或删除服务器而不影响生产者可用性，这样简化了部署和服务器管理的难度。

最后，部署多台工作者服务器会让系统容错性更好，并可以在某种程度上实现自我修复。如果有多台工作者机器，硬件失效可以被当作低优先级来处理。当后台服务器失效宕机时，整个应用不会崩溃，仅仅是吞吐量下降，而可用性并没有下降。异步任务降低的吞吐量对用户不可见，因此对用户来说没有什么影响。为了从失效中恢复，仅需要添加或者替换服务器，系统能自动自我修复并逐渐处理掉消息队列中的数据。

失效恢复和自我修复是水平伸缩系统的重要特性。

解耦

消息队列让我们实现系统高度解耦,这对系统架构有重要影响。本书第2章已经解释了了解耦的好处,但这里我想强调的是消息队列对提升解耦的影响。

正如本章前面提到的,使用消息代理可以隔离生产者和消费者。我们可以有多个生产者发布消息,有多个消费者处理消息,但它们从不互相直接交互。它们甚至不需要相互知道对方。

提示

当我们将两个组件分隔到它们互相不知道彼此存在的时候,我们就实现了高度的解耦。

理想状态下,我们创建发布者,它不关心谁消费及怎样消费消息。发布者需要知道的仅仅是消息格式和发布的目的地。另一方面,消费者也不需要知道谁发布了消息,为什么发布消息。消费者仅仅需要关注消息处理就可以了。图7-15所示为生产者和消费者怎样实现互不感知,它们彼此都不知道队列的另一头是什么。

通过消息中介代理实现这样高度的解耦使得独立开发生产者和消费者变得更容易,甚至可以由不同的团队,采用不同的技术进行开发。因为消息代理使用标准协议,所以消息本身可以采用标准的JSON或XML格式进行编码,这样消息代理将可以作为不同应用的集成点。

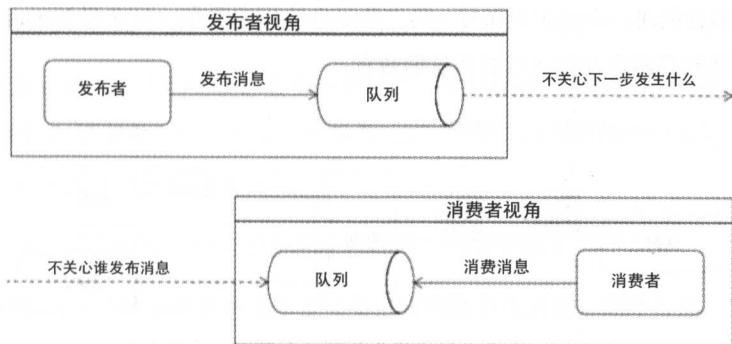


图 7-15 消息生产者和消费者解耦合隔离

提示

你可以将队列看做一个边界，边界另一头发生了什么是一个实现细节，这端代码并不知道。队列是你唯一的交互点，消息格式是规范。

虽然消息队列带来很多的好处，但记住没有银弹。接下来，我们将讨论消息技术的一些常见挑战。

消息队列相关的挑战

就像许多技术一样，消息技术有自己的一些挑战和代价。采用消息队列和异步处理常遇到的困难和陷阱包括消息无序、消息重新入队列、竞态条件、复杂度增加等。接下来，我们逐个详细分析。

消息无序

开发人员实现消息队列伸缩性时，面临的一个重要挑战就是消息顺序无法保证。这是因为消息是并行处理的，且消费者之间没有同步。每个消费者，每次处理一条消息，而不知道其他消费者同样正在以并行方式处理（这是一件好事）。既然消费者并行运行，当某个时候某个消费者变慢或崩溃，就会出现消息被无序消费的情况。

很难解释消息被无序消费，所以我们用序列图来展示。简单起见，我们看一个虚拟的例子，生产者向一个共享队列发送多种类型的消息。如图 7-16 所示，生产者发布两条消息，第一条消息创建一个新用户账号，第二条消息给用户发布一封欢迎 E-mail。注意，对同一个消息队列有两个并行运行的消息消费者。

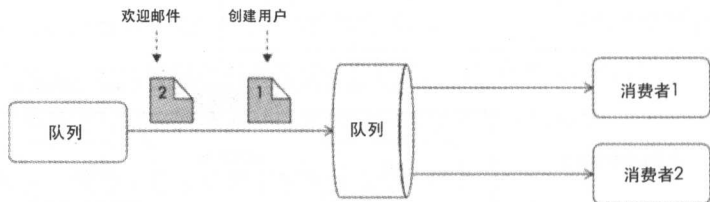


图 7-16 生产者创建两条与同一个用户相关的消息

每条消息被发送到其中一个消费者的机会相等，因为他们处在同一个逻辑队列。可以

设想这样一个场景，两条消息被分别发送到不同的消费者，如图 7-17 所示。现在，消息处理顺序取决于每个消费者有多快，以及任务 1 和任务 2 处理需要花多长时间。要么先创建账户，要么先发送 E-mail。这里有个显而易见的逻辑是，如果没有先创建用户账户，那么 E-mail 就不应该发送。这是竞态条件的一个经典例子，这两个没有同步的并行任务可能导致不正确的结果，因为它们对顺序有依赖。

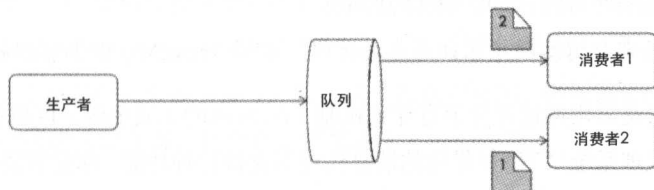


图 7-17 每个消费者收到其中一条消息

为了表现更糟糕的情况，我们看看另一个可能的失败场景，消费者 2 不可用或崩溃。在这个场景下，发送给消费者的消息可能被归还给队列，然后被重新发送给其他消费者。消息重新入队列是很多消息代理使用的策略，因为通常假定，在消费者通知处理完成前，消息有可能未被完全处理。图 7-18 所示为一条消息怎样重新入队列，以及它怎样被以错误的顺序发送给消费者 1。

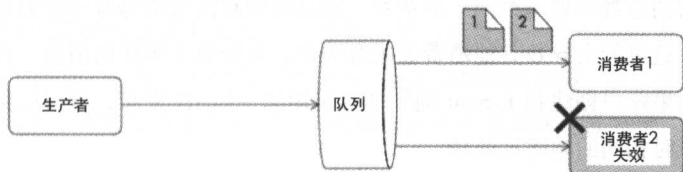


图 7-18 消费者失效导致消息被发送给另一个消费者

更令人担心的是，此场景还存在另一个困难。在失效前，消费者 2 不能保证消息确实未被处理。例如，消费者 2 已经发送了一封 E-mail，但就在给消息代理回发通知前崩溃了。这种情况下，消息 1 可能被处理两次。

幸好，我们还是有机会解决消息顺序问题的。通常有如下三种方法。

- 每个队列一个线程并限制消费者数量。有些消息队列保证顺序投递（先进先出），只要单个消费者每次消费 1 条消息就能保证消息顺序处理。不幸的是，此方案缺乏伸缩性，而且不是所有系统都支持。

- 构建一个不受消息顺序影响的系统。根据系统或需求的不同，这个方案可能简单也可能复杂，但看起来是目前最佳的方案。在前面的例子中，解决方法是改变谁来发，发哪条消息。前端应用只发布一条创建账户的消息，等账户创建成功后，消费者 1 会发送一条 E-mail 给用户。在这个场景下，消息顺序由应用流程保证。如果我们决定采用此方法，就需要确保所有工程师都理解这些约束。否则，不正确的消息顺序可能导致更糟糕的后果。
- 使用支持部分消息顺序保证的消息代理。采用 ActiveMQ 的消息组就是这种场景。

最好的办法是采用通过部分消息保证机制（ActiveMQ）或主题分区机制（Kafka）保证消息顺序的代理系统^[w52]。如果你的消息代理不支持这种功能，你就需要确保你的应用能按未知顺序处理消息。

部分消息顺序又称消息组（Message Group）是 ActiveMQ 提供的一个灵巧的机制。消息发送时带一个称为分组 ID 的标签。分组 ID 由应用开发者定义（例如，可以是消费者 ID）。同一组的所有消息保证按发布的顺序被消费。图 7-19 所示为不同组的消息怎样分割后发送给不同消费者。当第一个新的组 ID 消息被发布时，消息代理就将新的组 ID 映射给已存在的一个消费者。此后，所有属于同一组的消息都被发送给同一消费者。消息代理路由策略是基于映射而不是随机分发，这可能会导致其他消费者闲置等待，从而收不到任何消息。在这个例子中，如果账户创建和 E-mail 通知消息采用同一组 ID 发布，就可以保证它们按与发布时相同的顺序被处理。

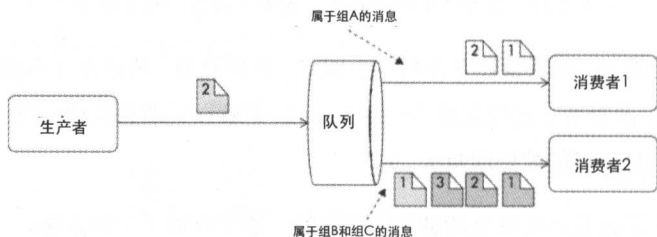


图 7-19 当第一条消息到达时消费者被分配到指定的消息组

当构建一个基于消息的应用时，消息顺序是一个需要考虑的重要问题。RabbitMQ、ActiveMQ 和 Amazon SQS 消息平台在并行工作者模式下无法保证全局消息顺序。事实上，

Amazon SQS 的消息顺序特别不可预知，因为它们的基础设施完全分布式，所以不支持消息顺序。你可以了解更多关于解决消息顺序问题的有趣方法^[w14,w52]。

消息重新入队列

正如前面提到的，在某些失败的场景下可能导致消息重新入队列。根据应用需求，处理此问题可能很简单，也可能很复杂。一个值得考虑的策略是：让应用本身能够接受至少发送一次而不是只能发送一次。如果允许消息可以多次发送给消费者，那么系统会更加健壮，消息队列及工作者的约束就会减少。要让这个方式起作用，必须保证所有消费者操作具有幂等性，这可能比较困难，甚至在某些场景下不可能做到。

幂等性消费者可以多次处理同一条消息而不影响最终结果。例如，设置价格为 55 美元就是一个幂等性操作。一个非幂等性的例子是给价格增加 5 美元。如果给价格增加 5 美元这个操作执行两次，就会导致合计增加 10 美元。此消息被处理两次会影响最终结果。相比之下，设置价格为 55 美元执行一次或两次，系统状态是相同的。

不幸的是，让所有消费者都满足幂等性并不容易。本质上，发送 E-mail 不是一个幂等性操作，因为给用户发送两封 E-mail 产生的结果与只发送一封 E-mail 并不相同。虽然这可以通过额外添加一个跟踪和持久层来解决，但又会增加复杂度，而且并不能处理所有的失败场景。相反，不管实践中是否可行，如果让消费者必须满足幂等性，可能会带来极高的代价。

最后，幂等性消费者对消息处理顺序更为敏感。例如，我们有两条消息，一条设置价格为 55 美元，另一个将同一产品设置为 60 美元，由于顺序不同可能会导致不同的结果。如果是两个非幂等性消费者，每个对价格增加 5 美元，那么可能对消息重新入队列（重发）比较敏感，但对顺序不敏感。

竞态条件可能性增大

异步系统的一个重大挑战是，传统编程模型中定义的操作顺序在异步条件下很可能发生错乱。因此，异步编程本身很难预测，更有可能发生竞态条件。因为任务被切分为更小

的部分，会产生更多的执行顺序可能性。

异步调用采用非阻塞方式，消息生产者执行时不用等待调用结果就可以继续执行。由于没有同步机制保障，不同消费者也可能以不同的顺序处理消息。异步系统的不同部分，尤其在分布式环境下，可能有不同的吞吐能力，导致整个系统内消息传播的延迟变得不平衡。

特别是在系统负载很高，或有节点失效等情况下，在系统内部分节点的代码执行可能会变得缓慢，这种情况也极可能产生不可预期的顺序。一些消费者获取消息可能比其他节点晚很多，从而产生很难复现的 BUG。

提示

你可以认为异步编程是没有调用栈的编程。事情^[w11]在能够执行的时候就执行，而不是一步步调用执行。

正如之前讨论的，竞态条件增加的风险主要由消息顺序问题引起。要保持严格审核代码的习惯，显式地排查竞态条件和无序处理的 BUG。这样可以降低问题产生的风险，使功能更加健壮。异步系统的状态越少越好。

复杂度风险

采用传统风格和面向消息编程风格开发的混合系统会更复杂，因为消息流并不是显式地被声明。对于生产者，它不知道消费者在哪里以及要做什么。而消费者无法确定消息在什么情况下被发布。随着系统增长和消息代码的添加，不考虑整体系统架构，要掌握各部分间的依赖变得越来越困难。

采用消息代理集成应用，必须经常采用文档记录依赖关系及主要的消息流。之前讨论过系统抽象级别及怎样在大脑中构建一个系统图像（第 2 章中涉及）。如果没有良好的文档来记录消息路由及整体系统的消息流程图，系统的复杂度会使开发者越来越难理解系统怎样工作。

简单起见，可以让文档基于代码自动创建。如果要想消息文档与代码保持一致，你必须想办法厘清系统中的依赖关系。

消息队列有关的反模式

除了挑战，你可能会强调一些设计的反模式。工程师一般会联想式思考，为近似的问题采用近似的解决方案。当解决方案被一次又一次证明是成功的，我们就称之为一个模式，但如果方案反复被证明难以维护伸缩，我们就称之为反模式。一个典型的反模式最初看起来很不错，但使用时间越长就会发现越多的问题。熟悉反模式，你就能在未来避免它们——就像对常见的设计缺陷产生疫苗一样。

将消息队列当作 TCP 套接字

一些消息代理允许你创建返回通道。返回通道提供了一种消费者回发消息给生产者的途径。不过，如果你大量使用这一功能，最后你的应用可能变成了同步而不是异步。理想状态下，你的消息应该是真正的单路请求（发射后不用管）。打开一个响应通道并等待响应消息，会使得消息组件紧密耦合，从而破坏消息机制带来的好处。响应通道也意味着消息代理两边的不同组件失效可能会互相影响。在构建可伸缩系统时，应该尽量避免返回通道，因为它们会导致同步处理和过度的资源消耗。

将消息队列当作数据库

你应该不会允许随机访问队列中的元素，也不会允许删除或修改消息，因为这会导致复杂度上升。消息队列最好被当作只能追加的流（FIFO）。我们可能经常看到这样的变形用法，比如消息队列构建在关系数据库或 NoSQL 引擎上，也就允许对消息进行二级索引及随机访问。使用随机访问修改和删除消息会让你无法实现水平伸缩及迁移到另一个不同的消息代理上。

如果你删除或修改了飞行中的消息（在队列中部的消息），就可能在以一种错误的方式使用消息队列。

耦合消息生产者和消费者

正如之前提到的，最好避免在生产者和消费者之间显式依赖。你不应该对类名进行硬编码或期望消息被一段特定的代码生产或消费。最好认为消息代理就是端点，消息内容就是协议，而不应该有其他假设和额外的必需的约束。在消息协议中没有显式声明的东西都

应该是实现细节，不应该影响协议的另一方。

例如，我见过的一个错误用法是将整个对象序列化并添加到消息体中。这意味着消费者也必须加载这个特别的类，如果不执行对象序列化代码就无法处理消息。更糟糕的是，消费者也必须采用与生产者相同的技术进行实现，程序部署时必须协调一致，以防止类不匹配。消息内部不能有逻辑或执行代码。消息应该是数据传输对象或能被生产者和消费者同时读写的字节序列。

将消息格式看做两方都能理解的协议，并禁止其他任何类型的耦合。

缺少坏消息处理

采用消息队列时，你必须使消费者代码能处理损坏的消息或者消息的 BUG。一个常见的反模式是总是假定消息是正确的。事实上，坏消息能够让消费者进程崩溃或以意想不到的方式失效。如果消息系统不能优雅地处理这些场景，你就应该冻结这个消息的处理，因为每一次消费者崩溃，消息代理就会将该消息重新入队列，并重发给另一个消费者，导致另一个消费者也崩溃。最后，所有消费者都崩溃。

为了防止这一点，你需要有计划地应对失效场景。你需要假定消息平台组件可能以未预料的方式崩溃、离线、熄火、失效；假定消息格式可能不正确甚至是恶意消息。否则，如果是构建系统的时候，想当然以为一切都以预期方式运行，这样的系统很容易变得不可用。

提示

抱有最好的希望，做最坏的准备。

根据消息代理的不同，你可以采用不同方式处理坏消息。在 ActiveMQ 中，你可以采用现成的死信队列策略。^[25]你需要做的就是设置消息限制数，它们会在失效指定次数后被自动从队列中删除。如果使用 Amazon SQS，则你可以使用近似发送计数器来处理坏消息。每当消息被重发，SQS 就增加它的近似计数器，这样你的应用就可以识别死消息，并把它们路由到死信队列或直接丢弃它们。类似地，在 RabbitMQ 中，你可以通过一个布尔标记消息是否被发送过，这可以被用来构造死信队列功能。不幸的是，这并不像使用计数器或现成的死信功能那样容易。

当使用消息队列时，你需要实现坏消息处理机制。

消息平台快速比较与选择

选择消息代理与选择数据库管理系统很相似。大部分消息代理可以满足大部分场景，但通常在使用前你需要知道自己要解决的问题。让我们快速浏览最常用的消息代理：亚马逊简单队列服务（SQS）、RabbitMQ 和 ActiveMQ。

一般说来，如果不了解应用场景很难推荐一个合适的消息平台，所以在最后决定前你需要做些研究。我推荐多去了解所选平台的细节。这里，我们先看看每个平台的特点和最合适的场景，在你做决定前先了解一些必要的知识。

亚马逊简单队列服务

SQS 以简单和可编程著称，一个亚马逊提供的云服务，它提供了支持大部分编程语言的应用编程接口（API）和软件开发工具（SDK）。它部署在亚马逊的服务器上，用户根据消息发布量和服务调用量按比例付费。

如果你使用 Amazon EC2 部署应用，Amazon SQS 当然值得考虑。使用 SQS 最主要的好处是你不用什么都自己管理。你不用考虑伸缩问题，不用额外雇佣专家，甚至不用考虑失效问题。你也不用为额外运行消息代理的虚拟服务器实例付费。SQS 能管理基础设施、可用性和伸缩性，保证消息任何时间都能发布和消费。

如果你在初创公司工作，遵循初创企业的行事方法，你就应该考虑引入 SQS 来提升你的优势。初创企业倡导最小化可行产品（MVP）开发和快速反馈回环。如果 SQS 的功能对你来说够用，那么使用 SQS 可以获得以下收益。

- 更快交付你的 MVP，因为 SQS 具有无设置、无配置、无维护、无意外的优点
- 关注产品和用户而不是花费时间在基础设施和技术难点上
- 使用 SQS 节省资金，无须自己管理消息代理

在早期开发阶段节省时间和金钱（开始的 6~12 个月）非常重要，因为你的初创公司可能会迅速改变方向。初创阶段的现实会让事情难以预测，也许在发布 MVP 后几个月，你意识到根本不需要消息组件，所有花费在上面的时间都是浪费！

如果你不给时间和金钱排优先等级,你的初创公司可能在找到适销对路的产品(给正确的人提供正确的服务)前就花光了钱。SQS 对初创公司来说通常非常合适,因为它的前期成本(时间和资金)最小。

提示

前期成本,无论是金钱还是时间都有可能成为浪费。变化的可能性越高,投资浪费的风险就越高。

为了说明 Amazon SQS 的竞争力,我们来看一个简单的成本比较。为了部署一个采用 ActiveMQ 或 RabbitMQ 的高可用消息代理,你至少需要两台服务器。如果使用 Amazon EC2,本书写作时,两台中等大小的实例需要花费每年 2000 美元。相比之下,如果使用 SQS,平均每条消息 4 个请求,每年你可以用相同的钱发布和处理 100 亿条消息。平均吞吐就是每秒 32 条消息。

此外,使用 SQS 可以节约自己开发、部署、管理、升级和配置消息代理的时间,这些花费很容易就是几千美元。假定初始设置和集成消息代理的时间花费只需要一周,外加一周的持续维护成本,最后还要至少每周花费 2 小时照顾消息代理而不是照顾你的客户和他们的需求。

简单说,如果你预计有很大的消息量,或你不知道预计何时,最好使用 SQS。SQS 只提供最基础的功能,因此即便你决定之后使用自己的消息代理,从 SQS 迁移也很容易。集成 SQS 时需要做的就是确保生产者和消费者没有直接与 SQS 的 SDK 代码耦合。推荐使用瘦包装器和你自己的接口,它们可以采用诸如依赖注入、工厂、门面和策略等设计模式来完成。^[1, 7, 10]图 7-20 所示为怎样通过使用 SQS 服务而不是定制的消息代理来简化基础设施。

说到伸缩性,SQS 表现得非常好。它根据你的需求自动伸缩,不需要特别准备或进行容量规划就能得到很好的吞吐量。可以支持你每秒每个队列生产消费几万条消息(假定有多个并发客户端);可以通过添加更多的队列、生产者和消费者进行无限制的伸缩。

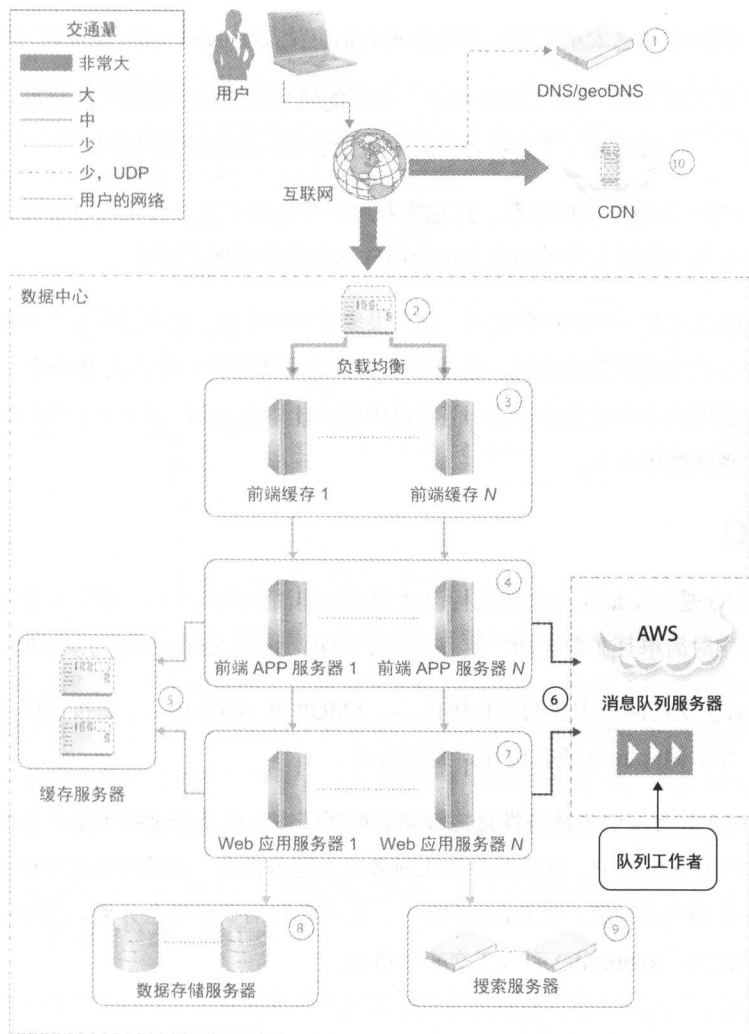


图 7-20 通过 Amazon SQS 简化架构

但是要记住 SQS 并不是万灵丹，它的伸缩性虽然很好但仍有一些限制。接下来，我们快速浏览它的缺点。

首先，Amazon 必须牺牲一些特性来确保 SQS 的伸缩性。SQS 缺失了如下特性：不提供复杂的路由机制；没有 RabbitMQ 或 ActiveMQ 灵活^[12, 25, L3]。如果你决定使用 SQS，你就不能在其中部署自己的业务逻辑或做任何修改，因为它是主机提供的服务。你要么按它本身的特性使用，要么不使用它。

第二，SQS 有消息大小限制。如果发布大消息（几十 KB）就可能被额外收费。

另一个要点是 SQS 消息不保证顺序，也可能偶尔会遇到消息重发的情况。即便只有一个生产者、一个队列和一个消费者，SQS 也没有任何消息顺序的保证。

最后，SQS 按服务调用收费，这意味着对不存在的消息进行轮询也会被记为服务调用，也意味着每秒发送几千条消息会比使用自己的消息代理更昂贵。

如果你的公司是一家稳固的公司，没有很多的不确定性，也许值得深入分析现有平台并选择一个自己管理的消息代理，它能带来更多的灵活性和更多的高级特性。尽管 SQS 从伸缩性和前期成本角度看很不错，但它提供的功能毕竟有限。接下来，我们看看自己管理的消息代理能提供什么。

RabbitMQ

RabbitMQ 是一个最初为金融机构设计的高性能消息队列平台。提供了很多有价值的特性，操作相对简单且非常灵活。实际上，正是灵活性让 RabbitMQ 与众不同。

RabbitMQ 支持两种主要的消息协议——AMQP 和 STOMP——是按通用目的设计的信息平台，并不偏向 Java 或其他任何编程语言。

RabbitMQ 最有吸引力的特性是能够动态配置路由，以及完全实现生产者与消费者解耦。在通常的消息机制中，消费者会和队列名或主题名耦合。这意味着在某种程度上系统的不同部分必须互相知道。在 RabbitMQ 中，生产者和消费者完全分离，因为它们与不同的端点类型交互。RabbitMQ 引入了交换的概念。

交换只是对端点的抽象命名，生产者用它对消息寻址。发布者不需要知道主题名或队列名，因为它们只向交换发送消息。另一方面，消费者从队列中获取消息。

生产者必须知道消息代理的地址，以及交换的名称，但不需要知道其他事情。一旦消息发布到交换，RabbitMQ 应用路由规则，并将消息复本发送给所有适合的队列。一旦消息出现在队列中，消费者就可以消费它们而不用知道交换的存在。

图 7-21 所示为 RabbitMQ 如何在物理上和逻辑上处理路由，如何隔离发布者和消费者。此处的技巧是路由规则可以在外部使用 Web 管理接口、AMQP 协议或 RabbitMQ 的 REST API 进行定义，也可以在发布者或消费者那里声明路由规则，但这不是必需的。你的路由配置可以在外部用一组独立的组件进行管理。

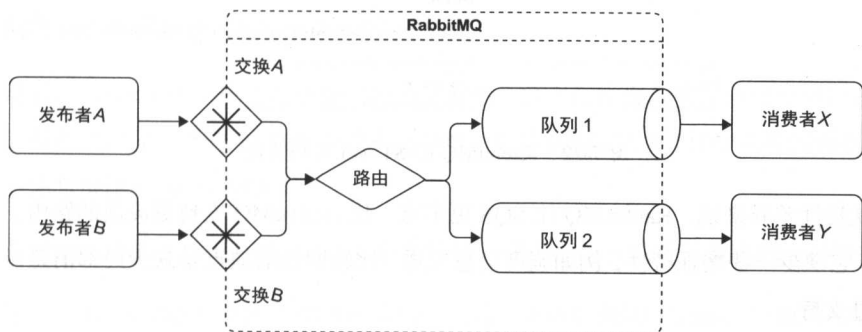


图 7-21 RabbitMQ 完全解耦生产者和消费者

如果你用这种方式思考消息路由，你就离面向服务的架构（SOA）更近了一步。在 SOA 中，你创建了若干高度解耦、自治和非常通用的服务，然后通过一些编排和策略，服务被组装为复杂应用。在 RabbitMQ 上下文中，你可以将其看做一个外部组件，能用它来决定系统的哪些部分可以相互通信，以及消息怎样在队列中流动。关于 RabbitMQ 路由的一个要点是：你可以远程改变路由规则；也可以在运行中修改，而不用重启任何组件。

值得注意的是，RabbitMQ 可以提供复杂路由，基于定制的路由键模式和更简单模式，例如直接队列发布，以及发布订阅。

使用 RabbitMQ 的另一个好处是你可以使用消息代理提供的远程 REST API 实现对其进行全面的配置、监视和控制。你可以用它创建任何内部资源，比如主机、节点、队列、交换、用户和路由规则。总的来说，你可以动态配置消息代理的任何特性而不用重启或在代理机器上运行任何定制代码。为了方便，RabbitMQ 提供的 REST API 结构组织和文档都非常好。图 7-22 所示为 RabbitMQ 自带文档的端点，因此你甚至不需要搜索对应版本的 API 文档。

GET	PUT	DELETE	POST	Path	Description
X				/api/overview	Various random bits of information that describe the whole system.
X				/api/connections	A list of all open connections.
X		X		/api/connections/name	An individual connection. DELETEing it will close the connection.
X				/api/channels	A list of all open channels.
X				/api/channels/channel	Details about an individual channel.
X				/api/exchanges	A list of all exchanges.
X				/api/exchanges/vhost	A list of all exchanges in a given virtual host.
X	X	X		/api/exchanges/vhost/name	An individual exchange. To PUT an exchange, you will need a body looking something like this: <pre>{"type":"direct","auto_delete":false,"durable":true,"arguments":[]}</pre>
X				/api/exchanges/vhost/name/bindings	A list of all bindings on a given exchange.
X				/api/queues	A list of all queues.
X				/api/queues/vhost	A list of all queues in a given virtual host.

图 7-22 RabbitMQ REST API 文档片段

就特性差异来说，RabbitMQ 比 SQS 更丰富，比 ActiveMQ 支持更灵活的路由。另一方面，它缺少一些增强特性，例如调度消息发送。比较明显的缺点是缺少局部消息排序和坏消息支持。

从伸缩性角度看，RabbitMQ 与 ActiveMQ 类似，性能也与 ActiveMQ 相当。它支持不同的集群和复制拓扑结构，但不幸的是，不提供现成的水平伸缩，你需要自己进行消息分区来实现水平伸缩。虽然不难，但也并不像使用 SQS 那么容易。

如果你的服务器不在 Amazon EC2 上或者你需要更大的灵活性，RabbitMQ 是一个不错的选择。如果你使用脚本语言例如 PHP、Python、Ruby 或 Node.JS，RabbitMQ 可以让你获得更多的灵活性，并利用 AMQP 和 RabbitMQ 的 REST API 在运行时动态调整配置。

ActiveMQ

最后，我要介绍一下 ActiveMQ。它的功能与 RabbitMQ 相似，并且具有相似的性能和伸缩能力。它们的主要区别是 ActiveMQ 采用 Java 编写，可以作为嵌入式消息代理与应用一起运行，而且如果你采用 Java 开发，这方面的优势有可能成为一个重要的决策因素。接下来，我们先浏览 ActiveMQ 的一些优势，然后讨论它的一些缺点。

它能在消息代理中运行应用代码，或者在应用进程中运行消息代理，使得你可以在消息代理两头重用同一份代码。而且可以获得更低的延迟，因为在同一个 Java 进程中发布消息基本上只是一次内存复制操作，比通过网络发送数据快了一个数量级。

ActiveMQ 不提供像 RabbitMQ 那样的高级路由，但你可以使用 Camel 实现同样复杂的特性。Camel 是一个用于实现企业集成模式的集成框架，^[10, 31-32] 是扩展 ActiveMQ 功能

的一个有力工具。Camel 允许你通过 XML 配置定义路由、过滤器、消息处理器，也允许你对不同组件集成自己的实现。如果决定使用 Camel，会使你的技术栈增加一项新技术，增加复杂度，不过也能使你获得更多高级消息特性。

除了基于 Java，ActiveMQ 还实现了一个称为 JMS（Java 消息服务）的通用消息接口，可以创建 Java 编写的插件。

最后，ActiveMQ 实现了之前提到的消息分组，这允许你部分确保消息有序传递。这个特性很独特，RabbitMQ 和 SQS 都没有此特性。如果你确实需要 FIFO 风格的消息机制，那可能你非得使用 ActiveMQ 不可了。

我们已经浏览了 ActiveMQ 的一些重要特性，接下来，我们看一看它的一些缺点。

第一，ActiveMQ 的路由不如 RabbitMQ 灵活。你可以使用 Camel，但如果你不是用 Java 开发，就可能给团队带来额外负担。而且，Camel 使用起来有些复杂，除非你的团队拥有经验丰富的工程师，否则我不推荐你使用它。还有一些特性允许你构建直接工作者队列和持久化的输出队列，但你还是不能基于复杂条件进行消息路由。

第二，与 RabbitMQ 相比，ActiveMQ 不能通过远程 API 实现完全控制。相比之下，RabbitMQ 可以通过 REST API 实现完全配置和监控。使用 ActiveMQ 时，你可以通过 JMX（Java 管理扩展）控制消息代理的某些方面，但如果你使用 Java 之外的编程语言开发，就不能用 JMX 了。

最后，ActiveMQ 发送大流量消息时，对峰值非常敏感。在对 ActiveMQ 进行压力测试时，我遇到过好几次，当持续压测一段时间后，在大流量消息冲击下，它就会直接崩溃。此外，因为它运行在 JVM 中，作为一个稳定的平台，ActiveMQ 也不能访问低层功能，例如内存分配和 I/O 控制。当你发布消息过多过快时，它有可能因内存不足导致消息代理崩溃。

最后的比较说明

根据 Google 趋势^[14]对比 RabbitMQ 和 ActiveMQ，我们可以看到 RabbitMQ 近年来更流行，并且两者都在增长。图 7-23 所示为两者过去 5 年的增长曲线。

产生这个趋势的一个可能原因是：RabbitMQ 被 Java 领域顶级主角之一的 SpringSource

收购，而 ActiveMQ 基于原来的程序重新开发，并起了新名字 Apollo。

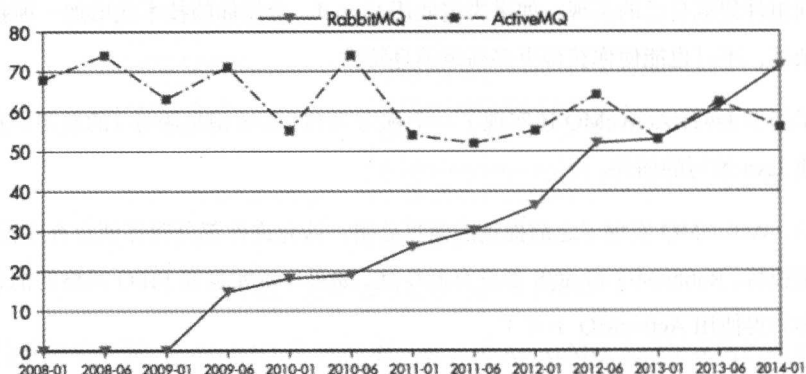


图 7-23 ActiveMQ 和 RabbitMQ 的搜索流行度 (Google Trends)

比较消息代理的另一种方式是看它们的可用性，以及它们如何应对极端条件。在这个比较中，ActiveMQ 在这三个系统中表现得最差。只要发送消息的速度快过其路由和持久化的速度，它就容易熄火甚至崩溃。ActiveMQ 在内存中缓冲消息，只要内存耗光它要么熄火要么完全崩溃。

RabbitMQ 在这样的场景下表现得更好，因为它有一个所谓后背压力的特性。如果消息发布速度比处理或持久化的速度快，RabbitMQ 就对生产者进行限流，以避免消息丢失及内存耗光。这种方法提高了稳定性和可靠性，但是因为当后背压力被触发时，消息发布速度会显著减慢，所以可能会导致发布者无法预知的延迟。

相比之下，SQS 表现得比 ActiveMQ 和 RabbitMQ 都要好，因为它支持很高的吞吐量，Amazon 会强制保证服务的高可用性。尽管 SQS 是一个托管平台，但在一些极端情况下仍可能遇到问题，你需要确保消息发布者能正确处理失效问题。你不需要担心消息代理崩溃、恢复过程或 SQS 的伸缩性，因为它们由 Amazon 管理。

无论你采用三种技术中的哪一种，吞吐量总是有限的，伸缩的最佳方式就是在多个消息代理实例间对消息进行分区 (SQS 是在队列间分区)。

如果你决定使用 SQS，你可以做到每秒每队列发送几万条消息，这对于很多创业公司足够了。如果你发现达到了极限，就需要创建多个队列实例，并通过消息分配实现整体吞吐量的扩展。由于 SQS 不保证消息顺序而且高级特性很少，因此在多个 SQS 队列间分

发消息其实就是随机选择一个队列并发布消息。至于消费者这一端，你需要相当数量的消费者来向每个队列订阅消息，并尽量保证消费者与服务器之间有相近的硬件配置，来提供均衡的消费能力。

如果你决定使用 ActiveMQ 或 RabbitMQ，每台机器的吞吐量会依赖很多因素。首先就是受每台机器可用 CPU 和内存（无论是独立主机还是虚拟服务器）的限制，其次是受平均消息大小、消息扇出率（队列数/每条消息被发送给多少消费者）的限制，也受是否持久化到硬盘的影响。无论单个消息代理实例每秒能处理多少消息，你的消息代理都要能够进行水平伸缩。

正如我之前提到的，ActiveMQ 和 RabbitMQ 自身都不支持水平伸缩，你需要实现应用级别的分区，即在多个代理实例上分发消息。你可以采用与第 5 章描述的应用层数据分区相似的方法。部署多个代理实例并分发消息，每个代理可以有相同的配置和相同的队列（或者交换和路由），以及一组专用消费者。

如果你使用 ActiveMQ 并依赖其消息分组特性实现局部消息有序，你就需要使用消息组 ID 作为一个分片键，这样所有消息就会发送到同一个消息代理，强制保证有序。如果你不需要消息保证有序，在发送消息时就可以随机选择消息代理，因为从发布者视角看，每个消息代理在功能上都是相同的。

消息平台很复杂，很难仅通过几页内容就分析出所有的差异和陷阱。正如之前讲过的，在做决策前你需要充分了解你的工具。在这部分，我只提到了最流行的消息平台，此外，还有很多其他消息代理可供选择。我相信消息技术是被低估了的，值得花时间去了解和學習。我推荐从阅读 RabbitMQ^[12]和 ActiveMQ 开始^[25]，也推荐读者看看 Kafka 的文章^[w52]。

事件驱动架构介绍

前面已经讲了很多东西，现在我还有一个更令人激动的概念要介绍：事件驱动架构（EDA）。这部分，我会解释传统编程模型和 EDA 之间的关键差异。我也会讲到它的好处，以及在一个较大的非 EDA 系统中如何使用 EDA。

首先，要理解 EDA，就不能再用软件中的请求响应术语来思考。相反，必须按组件执行即发生的思维来思考。交互思考的微小差异会对架构和伸缩性产生深刻的影响。我们

先来定义一些基本术语，并比较 EDA 和传统请求/响应模型的差异。

EDA 是一种架构风格，组件之间通过声明事件已发生进行交互，而不是发送执行请求。消费者一端，EDA 要对发生在系统内或系统外的事件进行响应。EDA 消费者不是一个服务，不为其他人提供服务，只对其他地方发生的事情做出响应。

事件是表示事情发生的对象或消息。例如，当在线上商店下订单时，事件就被声明或发送。在这个场景下，一个事件可能包含买家和商品信息。事件是一个实体，它持有一些数据用来描述发生了什么事情。它没有任何逻辑，相反，最好把事件看做一个数据，它描述了现实世界或应用中发生了什么。

截止现在，EDA 和消息之间的差异仍比较模糊。接下来，我们近距离看看以下几种交互模式之间的差异：请求/响应、消息和 EDA。

请求/响应交互

这是一种传统模型，类似传统编程语言如 C 或 Java 中的同步方法或函数调用。调用者发送一个请求，等待接收者处理消息并返回一个响应。我在本章前面详细描述了这种模型，所以这里不再赘述。有一个要点是调用者需要知道接收者在哪里，需要知道接收者的协议，并临时与接收者耦合起来。

时间耦合是同步调用的另一个术语，意思是如果接收者没有响应，调用者就无法继续执行。等待接收者执行完毕是耦合产生的来源。换句话说，这个调用栈中最弱的一环决定了整体延迟。（可以进一步阅读时间耦合相关资料。）

[w10, 31]

在请求/响应交互场景下，协议包含服务位置、请求消息定义和响应消息定义。客户端至少需要了解这些信息才能使用服务。我们在第 2 章讨论过隐含耦合，对于一个组件你需要知道的越多，你和它的耦合就越强。

直接队列交互

在这种交互模型中,调用者向队列或主题发布消息,由消费者响应。虽然这与事件驱动模型非常相似,但仍然给进一步耦合留下余地。在这种模型中,调用者可以向一个名为 `OrderProcessingQueue` 的队列发送一条消息,表明调用者知道下一步要做什么(订单需要被处理)。

这种方式的优点是它是异步处理,生产者和消费者之间没有耦合。不幸的是,通常生产者了解消费者,发送到队列的消息仍然是一个请求。如果生产者需要知道要做什么,那么它就与执行实际工作的服务产生了耦合。在协议上没有耦合,但逻辑上仍然耦合。

在基于队列的交互中,协议包含队列位置、发送到队列的消息定义及消息处理结果的预期。正如我已经提过的,我们已经减少了协议范围,因为响应消息不再是它的一部分。

事件驱动交互

最后,我们看看事件驱动交互模型。这里发布者不知道任何消费者。事件发布者创建一个事件实例,比如 `NewOrderCreated`,并将其声明到事件驱动框架中。框架使用企业服务总线(ESB),ESB 可以是内建组件,也可以使用消息代理比如 `RabbitMQ`。重点是发布事件不需要知道它的目的地,事件发布者不关心谁来响应或怎么响应。

就本质来说,所有事件驱动的交互都是异步的,发布者不需要知道消费者就能继续执行。

这种方式的优点是可以实现高度解耦,生产者和消费者不需要互相知道。一方面,既然事件框架将生产者和消费者绑在一起,生产者就不需要知道向哪儿发布事件,它只需要声明事件即可。另一方面,消费者不需要知道怎样获取到感兴趣的事件,它们仅需要声明感兴趣的事件类型,事件框架负责将事件路由到消费者。

需要指出的是,生产者和消费者之间的协议可以精简为事件消息定义。由于没有端点,因此不需要知道它们的位置。而且,发布者并不期望响应,因此协议也不包括生产者。发布者与消费者的所有连接点就是事件消息的格式和含义。

图 7-24 所示为在请求/响应交互模型中,客户端和服务怎样耦合在一起。它也展示了为了顺利工作,所有客户端和服务需要共享的信息。协议的总大小称作耦合表面积。

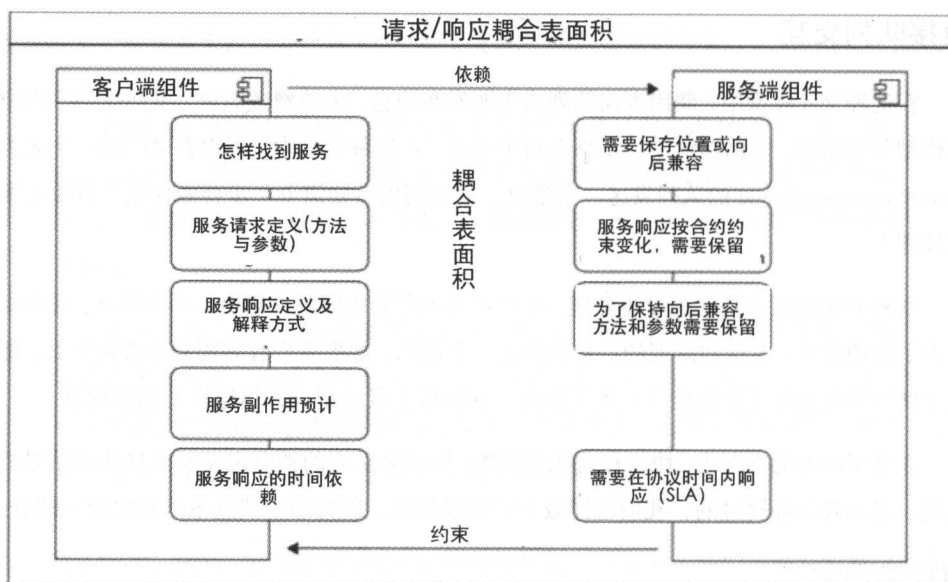


图 7-24 服务和客户端的耦合表面积

协议表面积用来度量耦合程度。为了协作，彼此需要知道的信息越多，耦合表面积就越大。这个术语来自图形化 UML 建模，两个组件的连线越多，它们的耦合程度就越强。

在请求响应交互模型中，客户端和服务有很多耦合的方式。它们需要能定位服务并理解它的消息。服务合约包含请求和响应消息。它们也在时间上耦合，因为客户端必须等待服务响应。最后，客户端经常要对服务方法做很多假定。例如，客户端的 `createUser` 方法可以假定用户对象在数据库某个地方被创建。

另一方面，根据协议设计的服务无法灵活适应不断变化的业务需求，因为通常需要保持协议的稳定性。在这种场景下，服务和客户端通过之前暴露的服务行为及请求响应消息耦合在一起。另外，服务也要支持 SLA（服务分层协议），这意味着必须快速响应，不能经常宕机。最后，服务受限于暴露给客户端的方式，这可能妨碍你为了更好的伸缩性将服务划分为较小的服务。

相比之下，图 7-25 所示为 EDA 交互。我们可以看到许多耦合因素已被移除，整体耦合表面积已经很小。组件之间不需要非常了解彼此，唯一的耦合仅仅是事件定义本身。发布者和消费者必须就事件内容和含义的理解达成一致。此外，事件消费者可能会受到事件消息的约束，因为某些特定数据没有包含在事件定义中，消费者需要询问共享信息源，否则它就无法访问任何信息。^[16-L7, 24]

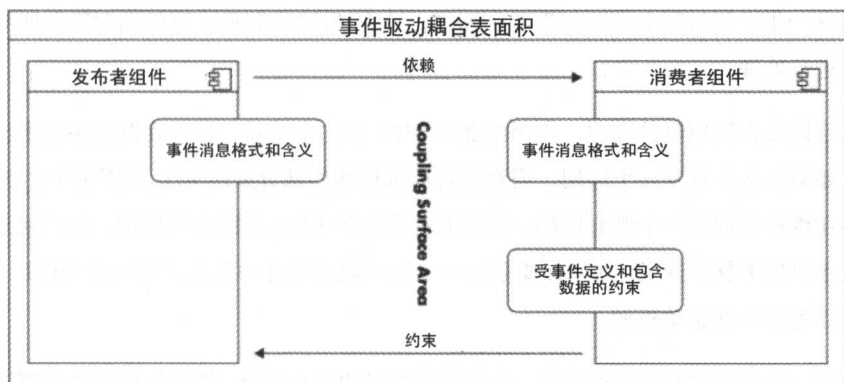


图 7-25 EDA 组件之间的耦合表面积

事件溯源是一种将应用状态的每个变化以事件形式持久化的技术。事件通常以事件日志文件或其他数据格式存储在磁盘上。同时，由事件消费者构成的应用会处理收到的事件。因此，系统可以恢复到一个旧的状态（例如每日快照）或重放事件到达同一状态。

我在实践中见过采用 EDA 架构的事件溯源技术，在该场景中有 150,000 并发客户端连接执行金融衍生品交易。如果服务器发生崩溃，整个系统通过回放事件日志恢复到最近一个状态。工程师也可以复制事件日志并在开发环境通过回放日志实现线上问题调试。非常酷。

事实上，分布式系统的异步复制经常以类似的方式进行。例如，MySQL 复制就采用了类似的方式，在 master 服务器上有变更后，数据修改被马上记录到 binlog 中。既然所有状态变化都在 binlog 中，slave 复制服务器可以通过回放 binlog 进行同步。^[16]唯一的差别是这些事件的消费者是复制服务器。将所有事件存储到日志中，意味着你随时可以添加

一个新的消费者来处理历史事件，这样看起来它好像从一开始就在运行。

事件溯源的一个重要限制是需要中心化状态和事件日志。重建应用状态仅仅依赖于事件日志本身，你可以用同样的顺序来处理它们。你可能要说，你需要从牛顿学说的观点对时间做假设，要求绝对对事件顺序和全局“现在”的概念。不幸的是，分布式系统在全局上存在跨度间隙，令顺序变得更困难，因为事件可能在世界不同地方不同服务器上同时发生。你可以进一步了解时间上的时间溯源和溯因，^[L7, 39]但简单起见，事件溯源需要所有事件按顺序进行处理。

无论你是否使用事件溯源，你仍然能从 EDA 架构中受益，甚至在既有系统中也可以使用。如果你从头开发一个应用，当然能自由选择哪些部分采用 EDA 风格进行开发，但即便你在维护和伸缩一个既有应用，仍然有许多场景 EDA 可以发挥作用，技巧就是用事件相关术语思考软件开发。如果你要添加一个新功能或组件并且不需要知道操作结果，那么你可以考虑事件驱动模型。

例如，你可以用传统方式开发一个在线购物车的核心部分，然后使用事件机制进行伸缩。通过发布事件，你可以让核心组件不再依赖外部组件，不破坏它的可用性和响应能力，并且可以通过添加新的事件消费者增加新特性。EDA 也能做到伸缩扩容，因为可以将不同的消费者部署到不同的服务器上。

小结

本章我们讲了许多内容，讨论了异步处理、消息、不同的消息代理和 EDA。要深入了解每个主题都需要专门一本书来展开讨论。我们的讨论相对简单和概括。异步处理与传统编程模型非常不同，但也非常值得学习。要记住的重要一点就是消息、EDA 和异步处理仅仅是工具。用来解决正确的问题非常有用，但如果用错了地方，就可能成为噩梦。

你可以基于伸缩性上下文进一步了解异步处理的价值，并获得足够的背景知识来自己探索这些主题。本章涉及的概念都很简单，但要完全了解这些概念背后的原理仍需要花费一些时间。同样的事物对不同的人需要用不同的方式进行解释，因此我强烈建议进一步阅读更多书籍^[31-32,24-27,12]和文章^[L6,w10-w11]。

异步处理仍然在持续发展。很多知名厂商像 VMWare (RabbitMQ, Spring AMQP)、LinkedIn (Kafka) 和 Twitter (Storm) 都参与进来。因为现在分布式系统构建方案各不相同, 像 Erlang 和 Node.js 这样的异步语言也越来越流行。带有事务、锁、同步处理的整体企业服务器看起来要逐渐步入历史。我们正进入一个轻量、创新和高并发技术的时代, 创业公司正持续投资这些解决方案。EDA 和异步处理正在经历复兴, 未来可能变得更流行, 因此每个工程师现在开始学习它们都是一个不错的投资。

8

数据搜索

构建数据、有效索引及执行复杂搜索是一个很有挑战的事情。随着数据从 GB 级别增长到 TB 级别，找到你要的数据变得越来越困难。当你读取、更新、删除，甚至插入新数据时，应用和数据存储需要执行搜索来准确定位到待读写的数据（或数据结构）。要更好地理解怎样搜索几十亿级别的数据，需要先熟悉索引如何工作。

索引介绍

在可伸缩网站中对数据进行有效索引是很一项重要的技术。即便你不想成为此领域的专家，你也需要对索引及搜索在持续增长的数据场景下如何工作有个基本的了解。

我们先用一个例子来解释索引和搜索如何工作。假定你有一个 10 亿用户的个人信息并需要快速搜索用户信息（我使用 10 亿数据是为了让伸缩性问题更明显，你在小数据集上也会面临类似问题），假设数据集包含姓名、E-mail 地址、性别、生日和用户账号（userID），那么这个数据结构如表 8-1 所示。

表 8-1 测试用户数据集的例子

User ID	名字	姓氏	E-mail	性别	生日
135	John	Doe	jdoe@example.com	男	10/23/86
70	Richard	Roe	richard@example.org	男	02/18/75
260	Marry	Moe	moemarry@example.info	女	01/15/74
...

如果数据没有做任何索引，就无法快速按条件找到用户，唯一的方式就是一行一行地扫描整个数据集。如果想在 10 亿用户中查询一个特定 E-mail 是否在数据库中，就需要执行多达 10 亿次比较。在最糟糕的场景中，如果一个用户不在数据集中，仍需要执行 10 亿次比较（只有检查了所有数据后才能确定用户不存在）。平均下来，也有可能花费 5 亿次比较来查询用户是否存在，因为有些用户离开始位置更近，有些用户离结束位置更近。

这种类型的搜索通常用术语“全表扫描”描述，因为需要扫描整个数据集来查找特定的数据行。可以想象，这种搜索的代价非常大。需要从磁盘加载所有数据到内存中来执行比较，并确认当前行是否就是要找的数据。全表扫描是搜索中最糟糕的案例场景，它的时间成本是 $O(n)$ 。

大写 O 是用来比较算法和估计成本的一种方式。用简单术语讲，大写 O 表示随着输入的变化，工作量有多大变化。其中 n 代表数据集的行数，大写 O 表达式用来估计在这个数据集上执行算法的成本。表达式 $O(n)$ 表示如果数据集大小加倍，那么算法执行的成本也会加倍。如果你看到表达式 $O(n^2)$ ，意思是如果数据集大小加倍，成本会按平方增长（比线性增长快很多）。

由于全表扫描的成本是线性的，因此用于大数据搜索并不是一个有效方式。通常，加快搜索的方式是在搜索的数据上建立索引。例如，如果你想基于 E-mail 地址搜索用户，则需要在 E-mail 地址字段上建立索引。

简单讲，你可以将索引看做一个查询数据结构，就像书的索引。要构建一本书的索引，需要对条目（关键字）按字母表顺序排序，并将它们映射到页码上。当读者查询指定术语对应的页时，可以在索引中快速定位并找到相应的页码。图 8-1 所示为数据是如何用图书索引的方式进行构造的。

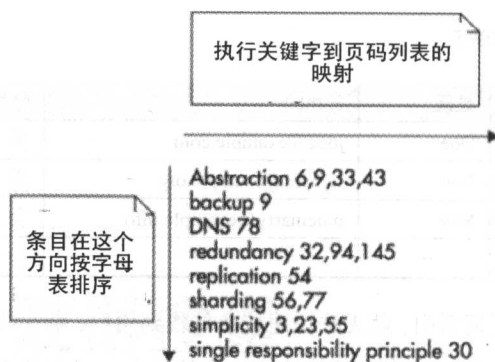


图 8-1 图书索引的结构

索引有两个重要属性。

- 索引是结构化的，是按特定顺序排序的，并为特定类型的搜索进行优化。例如，图书索引能回答诸如“某个关键词在哪些页？”之类的问题，但它不能回答类似“多个关键词构成的词组在哪些页？”的问题。虽然两个问题都指的是书中关键词的位置，但图书索引不能有效地回答第二个问题。
- 数据集在大小上减少了，因为索引比整个文本内容少很多，这样索引就能够被快速加载和处理。一本 400 页的书只有几页索引。这样搜索关键词就更快速，因为要搜索的内容很少。

大部分索引都是排序的，这是因为在排序数据集上进行搜索比在未排序数据上执行得更快。一个简单快速搜索算法的例子就是二分查找算法。当使用二分法查找时，不需要对数据集从头到尾进行扫描，可以跳过大部分比较项。算法取一个范围的排序数据，执行 4 个步骤就能找到数据。

1. 查看数据集中间项，判断它的值是相等、大于还是小于你要查找的值。
2. 如果当前值相等，你就找到了要查找的数据。
3. 如果当前值大于你要查找的值，可以继续较小项目中查找。然后回到第 1 步，数据集就降为一半。
4. 如果当前值小于你要查找的值，可以继续较大项目中查找。然后回到第 1 步，数据集也降为一半。

图 8-2 所示为二分法查找如何在一个排序数字序列上进行工作。可以看到，不需要比较所有的数字。

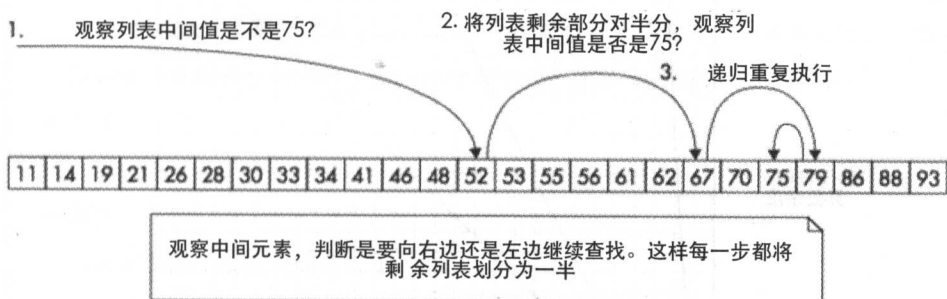
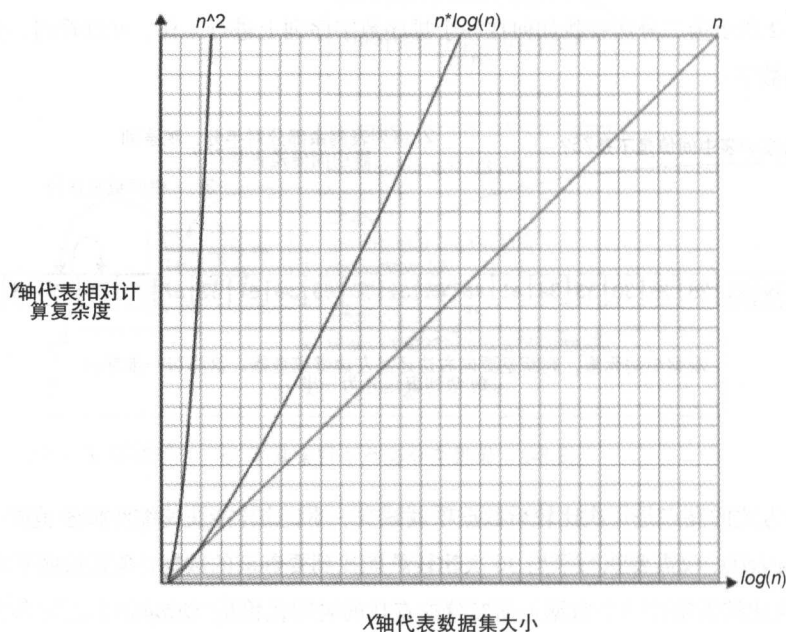


图 8-2 采用二分查找法搜索数字 75

这种方式的亮点是每次比较后数据项就减少一半，这样能快速缩小搜索范围。如果你有 10 亿 userID，仅需要执行平均 30 次比较就能找到要查询的数据。全表扫描平均需要花费 5 亿次比较来定位一个数据，而二分法查找的时间代价是 $O(\log n)$ ，这比全表扫描的 $O(n)$ 要少很多。

随着数据集的增长，算法和数据结构的选择越来越重要，因此很有必要熟悉一下符号 O 。一些常见的 O 符号表达式有 $O(n^2)$ 、 $O(n \cdot \log(n))$ 、 $O(n)$ 、 $O(\log(n))$ ，以及 $O(1)$ 。图 8-3 所示为这些复杂度的比较曲线，水平坐标是数据集大小，纵坐标展示了相对计算成本。可以看到， $O(n^2)$ 的计算复杂度增长非常快，导致很小的数据集计算量也很大。另一方面， $O(\log(n))$ 增长非常缓慢，甚至不会在图上注意到。与其他曲线相比， $O(\log(n))$ 更像常量时间 $O(1)$ ，使得在大数据集上的计算非常高效。

索引对搜索来说非常有用，但不幸的是，它需要增加一些开销。维护索引需要你用排序项来维护额外的数据结构，随着数据集的增长，这些数据结构会变得非常大，且代价也很高。在这个例子中，索引 10 亿 userID 需要使用高达 16GB 的数据。假设采用 64 位整数，就可能需要为每个 user 使用 8 字节存储 userID，8 字节存储数据偏移量。在这样的规模下，添加索引需要仔细考虑和规划，因为索引太大会消耗大量内存和 I/O（索引中存储的数据需要从磁盘读取并写入磁盘）。

图 8-3 复杂度 O 符号曲线

代价更高的是，将 E-mail 地址这样的文本字段进行索引需要更大的空间，因为被索引的数据要大于 8 字节。平均来说，E-mail 地址大约 20 字节，这样索引需要的空间就会更大。

考虑到索引增加了开销，就需要明确哪些数据值得索引，哪些不值得。为了做这些决定，你需要分析要执行的查询及每个字段的基数。

基数是特定字段值中唯一取值的个数。基数高的字段值得索引，因为这允许你将数据集减少到非常少的行数。

为了更好地解释如何估计基数，我们再看看之前的数据集例子。下面是所有带估计基数的字段。

● 性别

在大部分数据库中，一般只有两个性别，基数很小（基数为 2）。即便在一些数据库中有多个性别（例如变性男性），但整体基数仍然很小（最多几个）。

- 生日

假设用户大部分在 80 岁以下，10 岁以上，这样最多有 25,000 个唯一日期（基数大约 25,000）。假定用户分布不均匀，20 多岁的用户多于 70 多岁的用户，即便 25,000 个日期看起来很多，但是想想每天新出生的用户有几万或几十万。

- 名字

取决于出生的组合，可能有几万个不同的名字（基数大约几万）。

- 姓氏

与名字类似（基数大约几万）。

- E-mail 地址

如果 E-mail 地址用来唯一识别系统账户，那么其基数应该与总行数相等（基数=10 亿）。即便你没有要求 E-mail 的唯一性，其重复的也很少，这样基数非常高。

- User ID

既然 user ID 是唯一的，其基数也是 10 亿（基数=10 亿）。

低基数字段不适合用来做索引的原因是它们不能有效缩窄搜索范围。遍历了索引后，仍然剩余许多行要检查。图 8-4 所示为两个索引可视化排序列表。第一个索引包含 user ID 和在数据集中的位置。第二个索引包含每个用户的性别及在数据集中的位置。

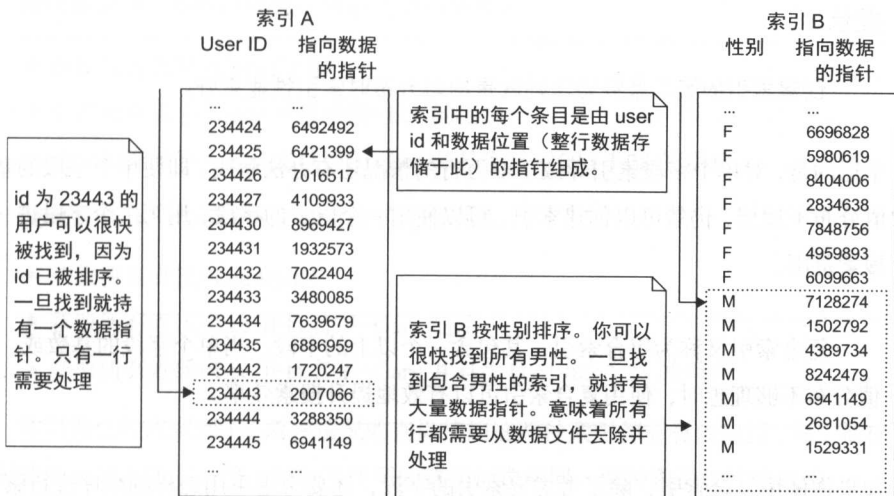


图 8-4 字段基数和索引有效性

两个索引都已经排序并按不同索引类型进行优化。索引 A 对基于 user ID 的用户查询进行了优化，索引 B 对基于性别用户查询进行了优化。

这里的关键点是在索引字段上搜索到一条数据的匹配速度非常快，因为搜索能跳过许多行，就像二分法查找算法一样。一旦找到了一个匹配行，就不能再有效地缩小搜索范围了，能做的就是一条一条地检查所有剩余行。在这个例子中，当使用索引 A 找到了一个匹配，就得到一行数据；相比之下，使用索引 B 找到一个匹配，会得到 5 亿行数据。

提示

在数据集上创建索引的第一条经验法则是基数越高，索引性能越好。

一个字段的基数并不是影响索引的唯一因素。另一个重要因素是数据项的分布。如果索引的字段的某些值仅在一个项中出现，而其他值在几百万条记录中存在，那么索引性能就会随着要查找的值而产生巨大的变化。例如，如果索引生日字段，那么最后可能形成一条用户分布的贝尔曲线。可能 1923 年 10 月 4 日出生的用户有 1 个，而出生于 1993 年 10 月 4 日的用户有 100 万个，那么搜索 1923 年 10 月 4 日出生的用户就能缩减到仅搜索 1 行，而搜索 1993 年 10 月 4 日出生的用户就会有 100 万剩余数据要检查和处理，这就使得索引的效率很低。

提示

创建索引的第二条经验法则就是均匀分布时索引性能更好。

幸运的是，对单个字段索引导致 100 万行的情况还有办法解决。即便单个字段的基数和取值分布不理想，仍然可以创建索引。可以使用一个以上的字段，用第二个字段进一步缩窄搜索范围。

复合索引又称为组合索引，其包含一个以上的字段。当单个字段的基数或值分布不够理想时，使用复合索引可以有效地提升搜索效率。

如果你使用复合索引，除了考虑要索引的字段，还要考虑采用怎样的顺序进行索引。当你创建一个复合索引时，就创建了一个按多个列进行排序的列表。就像你在 Excel 或

Google Docs 中按多列排序一样。最后数据排序的顺序会随复合索引中列的顺序的不同而不同。图 8-5 所示为两个索引：索引 A（索引了名字、姓氏和生日）和索引 B（索引了姓氏、名字和生日）。

这里关键的一点是索引 A 和索引 B 是为不同类型的查询做的优化。它们很相似，但不完全一样，你需要精确地了解哪种类型的索引更适合你的应用。

索引A				索引B			
名字	姓氏	生日	指向数据的指针	姓氏	名字	生日	指向数据的指针
CHARLES	LEE	12-Jan-1985	6492492	ANDERSON	LARRY	9-Jan-1984	8242479
DAVID	LEE	14-Aug-1984	6421399	ANDERSON	SCOTT	9-Mar-1978	2834638
DONALD	JACKSON	19-Mar-1986	7016517	ANDERSON	THOMAS	7-Nov-1985	7848756
FRANK	GARCIA	6-Dec-1978	8969427	GARCIA	FRANK	6-Dec-1978	8969427
FRANK	GARCIA	16-Jun-1981	1932573	GARCIA	FRANK	16-Jun-1981	1932573
FRANK	LEE	11-Sep-1980	6886959	GARCIA	GARY	8-Apr-1984	6099663
GARY	GARCIA	8-Apr-1984	6099663	GARCIA	GARY	7-Oct-1986	2007066
GARY	GARCIA	7-Oct-1986	2007066	JACKSON	DONALD	19-Mar-1986	7016517
GARY	LEE	28-Mar-1986	3480085	JACKSON	JOHN	2-Dec-1983	7639679
GARY	LEE	2-Sep-1986	6941149	JACKSON	MARK	7-Aug-1985	2691054
JAMES	THOMAS	22-Oct-1982	7022404	JACKSON	ROBERT	13-Dec-1983	8404006
JOHN	JACKSON	2-Dec-1983	7639679	LEE	CHARLES	12-Jan-1985	6492492
JOSEPH	THOMAS	3-Apr-1985	1720247	LEE	DAVID	14-Aug-1984	6421399
LARRY	ANDERSON	9-Jan-1984	8242479	LEE	FRANK	11-Sep-1980	6886959
LARRY	LEE	14-Apr-1981	4959893	LEE	GARY	28-Mar-1986	3480085
MARK	JACKSON	7-Aug-1985	2691054	LEE	GARY	2-Sep-1986	6941149
ROBERT	JACKSON	13-Dec-1983	8404006	LEE	LARRY	14-Apr-1981	4959893
SCOTT	ANDERSON	9-Mar-1978	2834638	THOMAS	JAMES	22-Oct-1982	7022404
THOMAS	ANDERSON	7-Nov-1985	7848756	THOMAS	JOSEPH	3-Apr-1985	1720247

图 8-5 复合索引中的列的顺序

使用索引 A，你可以快速得到以下查询结果。

- 获取所有名字=Gary 的用户
- 获取所有名字=Gary 并且姓氏=Lee 的用户
- 获取所有名字=Gary 并且姓氏=Lee 并且生日=1986-3-28 的用户

使用索引 B，你可以快速得到以下查询结果。

- 获取所有姓氏=Lee 的用户
- 获取所有姓氏=Lee 并且名字=Gary 的用户
- 获取所有姓氏=Lee 并且名字=Gary 并且生日=1986-3-28 的用户

你可能已经注意到了，两种情况的查询使用这两种索引都能有效执行。每个情况下的匹配值的顺序可能不一样，但它们找到的数据行数相同，两种索引的性能相当。

考虑另一个有趣的情况，尽管索引 A 和索引 B 都包含生日字段，但如果不知道名字

和姓氏就不能快速地搜索到 1984 年 4 月 8 日出生的用户。要通过索引 A 进行搜索，你还需要一个要查找的名字。类似地，如果要通过索引 B 进行搜索，你需要知道用户的姓氏。只有知道了最左边字段的精确值，才能通过提供第二列、第三列额外信息来缩小搜索范围。

理解这部分讲述的索引基本知识对设计和实现可伸缩的 Web 应用非常重要。特别是在使用 NoSQL 存储时，不能再认为数据是存储在库表中的，数据是按索引存储的。我们进一步分析这个理念，看看怎样通过优化数据模型实现快速数据访问，而不用关心数据集的大小。

数据建模

当使用 NoSQL 数据存储时，你要习惯将数据本身看做索引。

设计和构建可伸缩 Web 应用的一个主要挑战就是识别访问模式及基于这些访问模式对数据建模。数据正规化及从传统关系数据库中积累的简单经验法则在应对 TB 级别数据量时显得并不够用。超大数据集和数据存储的技术限制会要求你认真设计数据模型及思考这些数据关系上的用例。

为了伸缩数据层，你需要分析你的访问模式和用例，然后选择一种数据存储，再设计数据模型。更有挑战的是，你需要让数据模型足够灵活以便将来扩展。同时，你需要优化快速访问以便跟上数据量增长的速度。

这两种要求经常是矛盾的，优化数据模型通常会降低灵活性；反过来，增加灵活性又会导致性能和伸缩性的降低。接下来，我们将讨论 NoSQL 建模技术和具体的 NoSQL 数据模型案例来更好地阐释在实践中是如何做的，以及需要做哪些权衡。首先，我们看看 NoSQL 数据建模。

NoSQL 数据建模

如果你之前使用关系数据库，就会对数据建模非常熟悉。在设计关系数据库模式时，通常需要先分析数据。首先，可能要问自己“我需要存储的数据是什么？”，然后需要分析要持久化的数据内容，并做实体划分（数据库表）。一般需要考虑哪些信息要存储在哪些表中。也需要使用外键创建表间的关系。接着，会浏览整个模式设计，以便降低数据冗

余和循环关系。

经过这个过程，会得到一个正规化且灵活的数据库模式（数据对象集合），可以通过 SQL 查询来回答几乎任何类型的问题。结束之后，通常不再需要特别的功能，或者关心哪些功能用来执行哪些查询。数据库模式主要基于数据本身进行设计，而不是基于查询或用例。之后，随着应用被实现及各种查询的需要，会创建索引来加速这些查询。这些数据模式通常不会变，因为用它来应对任何类型的查询已经足够灵活。

不幸的是，基于数据的设计和正规化过程在应用到 NoSQL 数据存储时并不适用。NoSQL 数据存储需要数据模型更灵活（能够执行连接操作）且更具伸缩性，因此你需要用不同的方式来应对。

为了在 NoSQL 数据存储中设计更好的数据模型实现更有效的访问，需要改变设计数据模式的方式。不是一开始就思考数据本身，而是从查询开始考虑。我认为在 NoSQL 世界设计数据模型比在关系数据库下设计要难得多。一旦为特定类型的查询优化了数据模型，就不能执行其他类型的查询。设计 NoSQL 数据模型更多的是关于权衡和数据布局优化的，而不是数据正规化的。

当为 NoSQL 数据存储设计数据模型时，需要首先识别所有的查询和访问模式。只有当你理解了数据怎样被访问，你才能继续识别数据的关键内容并找到将其结构化的方式，这样你才能有效地执行所有的查询类型。

例如，你正在基于关系数据库设计一个电子商务网站，可能没有仔细考虑数据如何被查询。可能你会对产品表、产品类目表，以及产品评价表进行拆分，如图 8-6 所示。

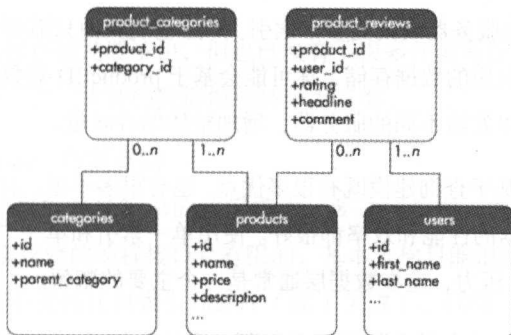


图 8-6 关系数据模型

如果你在设计同样类型的电子商务网站,并需要执行每小时百万级用户对百万级商品的访问,那么你可能会决定使用 NoSQL 存储来扩展系统。你应围绕主要查询进行数据建模,而不是用通常的正规化模型。

例如,你最重要的查询是加载产品页,它用来展示产品相关的元数据,比如名字、图片 URL、价格、所属类目,以及平均用户评级,你会为这个用例优化数据模型,而不是让它保持正规化。图 8-7 所示为一个备选数据模型,每个集合中有数据文档范例。

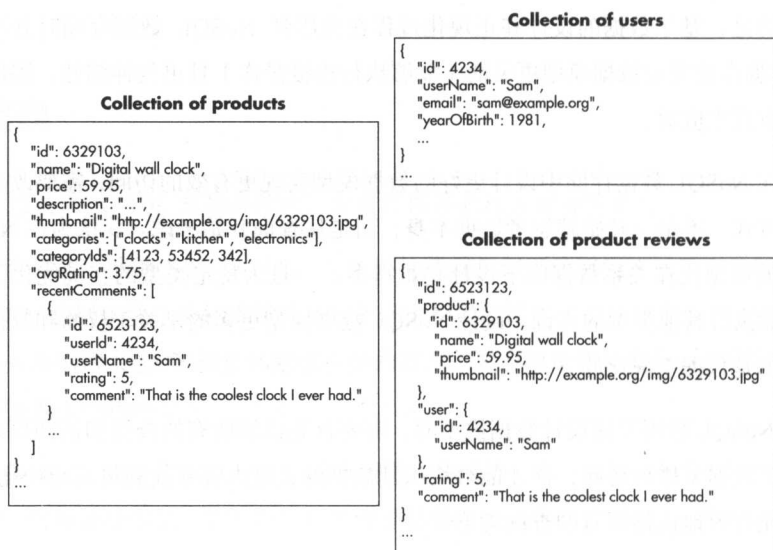


图 8-7 非关系数据模型

将大部分数据组合到产品实体,就能通过一次文档访问请求到所有需要的数据。不需要做表连接、查询多台服务器或浏览多个索引。产品呈现页面只需要一次索引查询获取一份文档数据。根据你采用的数据存储,你可能会基于 productID 对数据进行分片,这样涉及不同产品的查询可以发给不同的服务器,增加整体的吞吐量。

数据反正规化和基于查询建模既有很多优点,也有很多不足。主要的收益就是无论数据集有多大,访问数据的性能和效率都很好。使用单一索引和单一“表”使得磁盘操作得以最小化,降低了 IO 压力,这在数据层通常是一个主要的瓶颈。

另一方面,反正规化会带来数据冗余问题。在这个例子中,正规化模型(SQL 表)中,类目放在一张独立的表中,每个产品通过 product_categories 表被连接到类目。这样,

category 元数据只存储一次,产品元数据也只存储一次(product_categories 表仅包含引用)。在反规范化方式中(类似 NoSQL),每个产品有一个嵌入的类目列表。类目并不独立存在,它们是产品元数据的一部分。这会导致数据冗余,更重要的是,更新数据变得更困难。如果你需要改变产品类目名,你需要更新所有属于这个类目的产品,因为类目名存放于每个产品信息中。这会导致极大的成本,尤其当没有索引让你查找属于某个类目的所有产品时。在这个场景下,要更新类目名,需要做一个全表扫描并检查所有产品。

提示

灵活性是优秀架构的一个重要属性。再引用一下 Robert C. Martin 的话,“优秀架构使得未定决策最大化。”你需要权衡数据反规范化和特定访问模式的优化。为了性能和伸缩性牺牲一些灵活性。这些权衡考虑非常关键,需要认真分析和应对。

可以看到,反规范化是一把双刃剑。它能帮我们优化和伸缩,也可能带来很多限制,让未来的变更变得非常困难,并可能产生某些无法有效搜索数据的情况而不得不执行全表扫描,还可能导致一些情况使你不得不创建很多额外表,添加更多冗余来加快数据访问效率。

不考虑其缺点,当你使用 NoSQL 数据存储时,需要逐渐习惯基于数据访问模式和用例建模。就像第 5 章提到的, NoSQL 数据存储比关系数据库更特殊,需要不同的数据模型。总之, NoSQL 数据存储不支持连接,需要基于访问模式而不是数据本身进行归组和索引。

尽管 NoSQL 数据存储发展不快,但也已经有了很多开源项目,使用最多的 NoSQL 数据存储总体上可以按它们支持的数据模型被分为三类。

• 键值 (key-value) 数据存储

这些数据存储仅支持最简单的访问模式。要访问数据,需要提供数据存储的键(key)。键值存储的编程接口是有限的,基本上你只能根据 key 来设置或获取对象。键值存储不支持任何索引和排序(除了主键)。同时,它的复杂性最小,可以基于键实现自动分片,因为每个键是独立的,访问值的唯一方式就是提供键。

对于一对一的查询来说，键值存储非常好，但当你需要排序的对象列表或需要建立对象间关系时就不太适用。键值存储的产品有 Dynamo 和 Riak。Memcached 也是一种形式的键值数据存储，但它并不支持持久化数据，这更像是一个缓存而不是存储。另一个经常使用的键值数据存储是 Redis，不过它提供比键值映射更丰富的数据结构。

● 宽列数据存储

这些数据存储按照复合索引对数据建模。数据建模在这里仍是一个挑战，因为它与关系数据库很不同，但它更具实用性，因为它构建了排序列表。在宽列存储中没有连接的概念，因此反正规化是一个标准做法，但反过来宽列伸缩性很好。它们通常提供现成的数据分区和水平伸缩。对于超大数据集，例如用户生成的内容、事件流、传感器数据等是一个不错的选择。宽列存储的产品有 BigTable、Cassandra 和 HBase。

● 面向文档的数据存储

这些数据存储允许你构建更复杂的对象和索引。基于文档的数据存储在数据模型上使用文档概念作为最基本的数据块。文档数据结构包含数组、映射和嵌套数据结构，比如 JSON 或 XML 文档。文档有灵活的模式（基于每个文档可以灵活添加删除字段），文档数据存储通常允许添加更复杂的索引。文档存储一般提供一个富数据模型，它通常适用于这样一些场景，比如很难将预定义模式应用到数据模型（很难创建类似 SQL 的正规化模型），而同时又要考虑伸缩性。面向文档的数据库有 MongoDB、CouchDB，以及 CouchBase。

还有其他类型的 NoSQL 数据存储，例如图数据库和对象存储，但它们不太流行。详细探讨这些数据存储超出了本书的范围，尤其是因为 NoSQL 数据存储市场非常碎片化，每种数据存储都朝不同方向逐渐发展来满足特定的需求。

我们暂时不需要讨论所有的数据存储，首先可以看一些数据模型的例子，从而了解实践中的 NoSQL。

宽列数据存储的例子

假定一个场景——构建一个类似 eBay 的在线拍卖网站。如果你要使用关系数据库方法构建数据模型，就需要先寻找实体及数据正规化。就像我之前提到的，在 NoSQL 的世

界，你需要先分析你要执行什么样的查询，而不只是你要存储什么数据。

假定你有以下场景需要满足：

1. 用户注册和登录。
2. 登录用户能通过拍卖页面查看拍卖详情。
3. 商品拍卖页包含产品信息，例如标题、描述和图片。
4. 商品拍卖页还包括竞拍列表，含有竞拍的用户名及竞拍价格。
5. 用户要能修改名字。
6. 用户能通过查看其他用户的资料页查看他们的竞拍历史。用户资料页包含用户详情，例如用户名和信用分值。
7. 用户详情页包含用户竞拍的商品列表。每个竞拍展示了商品名，一个到商品竞拍页的链接，一个用户竞拍价格。
8. 系统需要支持几十亿用户及几亿商品，每个商品有几百万竞拍。

分析过用例后，你可能会决定使用宽列存储例如 Cassandra。Cassandra 能让你在高可用和自动水平伸缩能力之间做一个平衡。你需要对这些场景进行建模，并满足这些业务需求。

Cassandra 数据模型一般展示为表，其行数无限，列数也几乎无限，每行可以有不同列，列名可以随时决定（没有表定义、约束模式，列可以在添加行时动态创建）。

每行有一个行键作为主键，同时有一个表的分区键。行键是一个字符串——通常唯一标识一行数据，并被 Cassandra 索引。基于行键，行数据被分布在不同的服务器上，因此需要一次读取的数据必须存放在一行中。图 8-8 所示为行数据按行键进行索引，列数据按照列名进行索引。

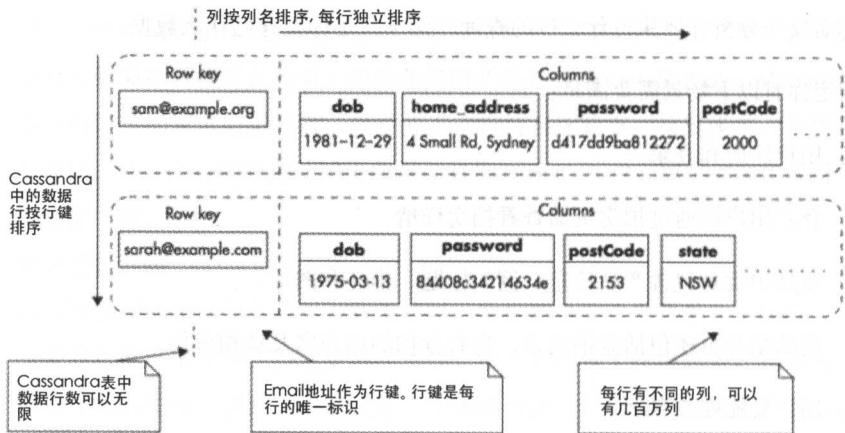


图 8-8 Cassandra 表中的两行数据片段

Cassandra 在表中组织和排序数据的方式与复合索引的工作方式类似。当要访问数据时, 需要提供行键和列名, 它们都被索引了。因为列是排序后存储的, 你可以在列上执行快速扫描来获取邻近列。由于每一行独立存在并且没有表模式定义, 因此不能快速地基于特定列值查询多行数据。

你可以将 Cassandra 表可视化为一个复合索引。图 8-9 所示为怎样在类似 MySQL 的关系数据库中定义这个模型。索引将行键作为第一个字段, 列名作为第二个字段。值对象包含了 Cassandra 字段的数据值, 因此不需要从其他地方加载数据。

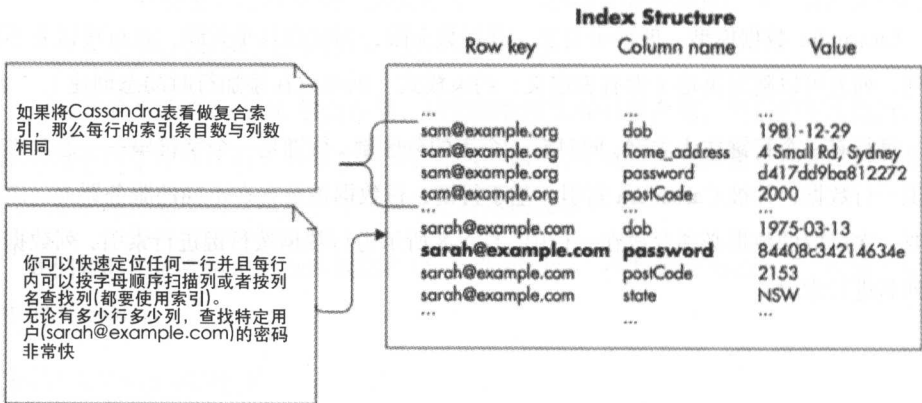


图 8-9 Cassandra 表看起来像复合索引

就像之前提到的，索引结构是按照字段进行优化的，以便快速执行访问。当你基于行键和列名访问数据，Cassandra 能快速定位数据，因为它遍历索引而不需要执行任何表扫描。这种方式的一个问题是那些不查找特定行键和列的查询性能会很低下，因为此类查询需要代价高昂的扫描操作。

提示

许多 NoSQL 数据建模技术最终都归结为复合索引。因此，使用索引的查询效率会很高，不使用索引的查询会执行全表扫描。

回顾在线拍卖网站的例子，你可以在 Cassandra 中创建用户表。你可以用用户 E-mail 地址（或用户名）作为用户表的行键，用户登录后能更有效地找到用户。为了确保总是能得到行键及用户能够持续访问，可以将其存放在 HTTP 会话或加密 cookie 中。

然后可以存储用户属性到用户表的不同列，例如名字、姓氏、手机号、哈希密码等。由于没有预定义模式，一些用户可能有额外列，比如账单地址、收货地址或者最佳联络方式等。

提示

每当需要查询用户表时，你可以针对指定用户查询，这样可以避免代价很高的表扫描。之前提过，Cassandra 表就像复合索引。行键是索引的第一列，因此你通常需要提供行键以便快速检索数据。你也可以提供列名或列名范围查找单独的属性。列名是复合索引的第二列，提供列名可以进一步提升搜索速度。

类似地，还可以对拍卖商品进行建模。每个商品被行键即 ID 唯一标识。列会展示商品属性，比如标题、图片 URL、描述、分类等。图 8-10 所示为用户表和商品表看起来的样子。通过这两张表，你可以通过行键快速查找任何商品和用户。

为了满足更多用例，你可能需要存储哪些用户竞拍了哪些商品之类的信息。为了有效执行这些查询，你可能需要为以下两种访问模式的优化存储一些信息：

- 获取某个商品的所有竞拍（用例 4）
- 获取某个用户的所有竞拍（用例 7）

用户表样本行					
Row key		Columns			
sam@example.org		dob	homeAddress	password	postCode
		1981-12-29	4 Small Rd, Sydney	d417dd9ba812272	2000

商品表样本行					
Row key		Columns			
345632		classifications	description	image1	image2
		4232,12,55,123	Have you ever ...	http://example..	http://example..
					title
					Digital wall clock

图 8-10 用户和商品表

为了执行这两种访问模式，你需要创建两个额外索引：一个按商品 ID 对竞拍进行索引，另一个按用户 ID 对竞拍进行索引。写此文的时候，Cassandra 还不支持对所选列进行索引，因此你需要创建两个额外 Cassandra 表来提供这些索引。你可以创建表 `item_bids` 存储每个商品的竞拍，第二个表 `user_bids` 来存储每个用户的竞拍。

此外，你还可以使用 Cassandra 的另一个特性“列族”来避免创建额外表。使用列族，你最后仍可以使用反正规化的和重复的数据。简单起见，这里先使用独立表做例子。每当用户竞拍，Web 应用需要写这两张表来保证两个“索引”同步。幸运的是，Cassandra 为写操作做了优化，从伸缩性和性能角度看，写多张表并不是问题。

图 8-11 所示为这两张表的样子。如果再仔细看 `user_bids` 表，你可能会注意到列名包含了时间戳和商品 ID。使用这个技巧，可以按时间排序存储竞拍信息，并将它们按时间顺序展示在用户资料页上。

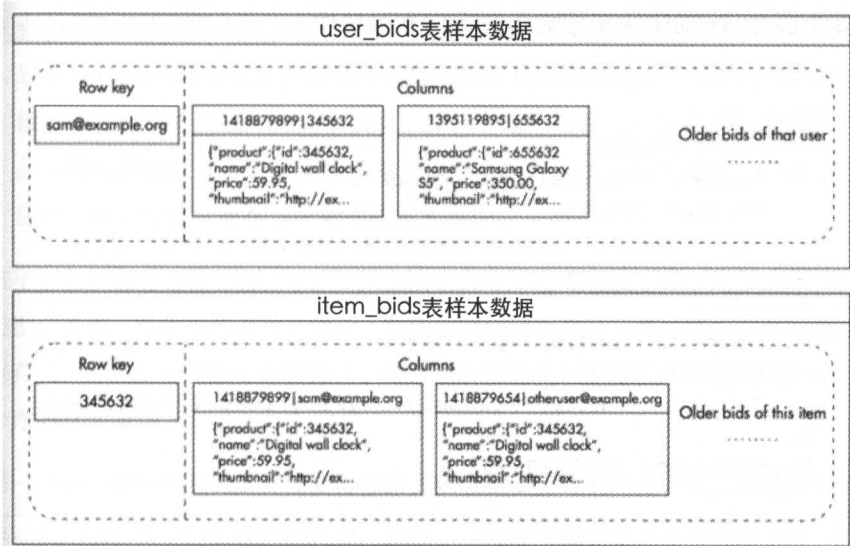


图 8-11 基于用户和商品的竞拍表

通过这种方式存储数据可以让你很快地将数据写入这些表。每当你需要竞拍时，就可以序列化竞拍数据并执行以下两个命令。

- 在 user_bids 表中设置行键为 “\$user_email”，列名为 “\$time|\$item_id” 的数据
- 在 item_bids 表中设置行键为 “\$item_id”，列名为 “\$time|\$user_email” 的数据

Cassandra 是一个最终一致性存储，因此这样执行写操作可以确保你不会丢失任何写操作。即便一些写操作延迟了，它们最后在所有服务器上仍然是相同的。此外，每台服务器上的命令执行顺序互不相关，你可以将同一条命令执行多次而不影响最后的结果（让这些结果变得幂等性）。

这里的竞拍数据并不值得反正规化和冗余化，代码清单 8-1 展示了这一点。你可以在 user_bids 和 item_bids 表中同时设置相同的数据。序列化的竞拍数据会包含足够的产品信息和竞拍用户信息，这样在用户资料页或商品详情页展示的时候不用从其他表中获取额外的信息。数据反正规化可以让你仅用一次 item 表查询和一次 item_bids 表列扫描就呈现商品信息页。用类似的方式，你可以通过一次 users 表查询和一次 user_bids 表列扫描就呈现用户资料页。

代码清单 8-1 序列化的竞拍数据被存储在列值中

```
{
  "product": {
    "id": 345632,
    "name": "Digital wall clock",
    "price": 59.95,
    "thumbnail": "http://example.org/img/6329103.jpg"
  },
  "user": {
    "email": "sam@example.org",
    "name": "Sam",
    "avatar": "http://example.org/img/fe6e3424rwe.jpg"
  },
  "timestamp": 1418879899
}
```

数据模型是否设计完成，很重要的一点就是用已知用例对其进行验证。用这种方式可以确保数据模型支持所有必需的访问模式。在这个例子中，你可以浏览以下用例：

- 创建一个账户并登录（用例 1），可以使用 E-mail 地址作为行键来快速定位用户行数据。用同样的方式，可以检测一个给定的 E-mail 地址的用户是否存在。
- 加载商品拍卖页（用例 2、3、4）可以根据 ID 查询 item，从 item_bids 表加载最近的竞拍数据来实现。Cassandra 允许从指定列名开始获取多列，因此竞拍数据可以按时间顺序加载。每个商品竞拍数据包含所有必要的信息以便呈现页面片段，而且不需要额外的查询。
- 加载用户页（用例 6、7）以类似的方式工作。可以从 users 表基于 E-mail 地址获取用户元数据，从用户竞拍表获取最近的竞拍数据。
- 更新用户名（用例 5）。这是一个有趣的场景，因为用户名存储在所有竞拍数据中，包括 user_bids 表和 item_bids 表。更新用户名必须是一个离线过程，因为它需要修改更多的数据。当用户要修改 user name 时，需要将一个作业添加到队列中，并将其执行推迟为一个离线过程。可以通过 user_bids 表查找某个用户所有的竞拍数据。然后，需要加载这些竞拍数据，反序列化，修改其中的 user name 然后再保存回去。通过从 user_bids 表加载每条竞拍数据，可以看到它的 timestamp 和 item ID。类似地，可以执行一个额外的 SET 命令来修改 item_bids 表中相同的竞拍元数据。

- 存储几十亿的竞拍数据，几亿的用户数据，上百万的商品数据（用例 8），这些场景通过 Cassandra 提供的自动分片功能和行键选择功能实现。用 user ID 和商品 ID 做行键，可以将数据分区到小块中，并均匀地分布在大量服务器集群中。拍卖商品不会收到超过一百万次竞拍，用户也不会执行几千上万次竞拍。因此，数据就能够分区并有效分布在尽可能多的服务器中。

这里有些权衡考虑需要注意。以“复合索引”的形式构造数据可以让你快速执行一些特定类型的查询。将竞拍数据反正规化，性能伸缩性得以提升。通过将所有竞拍数据序列化并保存为一个值，可以避免连接操作，因为竞拍页区块中要展示的所有竞拍数据都已经在一个序列化对象中了。

另一方面，竞拍数据反正规化使得修改冗余数据更困难且耗时。将数据按索引形式进行构造，你可以实现对特定查询的优化。这样，一些特定查询性能表现得相当好，但其他查询性能却很差。

找到一个灵活的数据模型来支持所有访问模式并提供最大的灵活性是 NoSQL 面临的真正挑战。例如，使用这个例子中的数据模型，使用竞拍的最高价格来查找商品性能表现并不好。由于没有索引，无法快速找到这个数据，因而需要执行全表扫描才能获得正确的结果。更糟糕的是，给 Cassandra 表添加索引并不容易。需要添加新的列或表进一步反正规化。

解决 NoSQL 索引问题的另一种方式是为复杂查询使用专用搜索引擎，而不是使用单一数据存储来满足所有用例。接下来，我们快速浏览搜索引擎，看看它是如何完善可伸缩 Web 应用数据层的。

搜索引擎

现在，几乎所有的 Web 应用都需要执行复杂的搜索查询。例如，电子商务平台允许用户在搜索产品信息时使用任意的查询条件组合，例如类目、价格范围、品牌、库存、位置等。难度更大的场景是，用户使用任意单词或短语来搜索，并需要按自己的偏好排序。

无论你是否使用关系数据库或 NoSQL 数据存储，即便使用最好的模型和最好的技术，在大数据集上执行灵活搜索都是一个巨大的挑战。

要让用户执行大范围的查询要么需要构建大量专门为特定场景优化的索引,要么需要使用专用搜索引擎。在决定是否需要使用专用搜索引擎之前,我们先快速介绍搜索引擎,以便更好地理解它能做什么及怎样做。

搜索引擎介绍

你可以将搜索引擎看做专门用于搜索文本或其他数据类型的数据存储。与关系数据库或 NoSQL 数据存储相比,它做了不一样的权衡考虑。例如,一致性和写性能相比快速执行复杂查询来说就不那么重要。由于要为特定交互模式做优化,它对内存消耗和 I/O 吞吐的需求又有所不同。

在开始使用专用搜索引擎之前,需要先理解全文本搜索是怎样工作的。全文本搜索及现代搜索引擎背后的核心概念是倒排索引。

倒排索引是一种用来搜索短语和单词的索引(全文本搜索)。

迄今为止,我们讨论过的索引类型要么是精确值匹配搜索,要么是前缀搜索。例如,假定基于一个包含电影标题的文本字段构建了一个索引,使用标题“*It's a Wonderful Life*”,你可以快速找到对应的数据。有些索引类型允许你使用前缀为“*It's a Wonderful*”的所有标题来执行快速搜索,但它不允许在一个文本框中搜索单独的词。如果用户输入了“*Wonderful Life*”,他不会找到包含“*It's a Wonderful Life*”的记录,除非使用全文本搜索(使用倒排索引)。倒排索引允许搜索文本框中的任何单词,无论它的位置和顺序是什么。例如,搜索“*Wonderful Life*”或“*It's aLife*”也能找到“*It's a Wonderful Life*”的记录。

当采用倒排索引对“*The Silence of the Lambs*”(沉默的羔羊)这样的文本进行了索引,它先被划分为符号(比如单词)。然后每个符号被预处理以提高搜索性能。例如,所有单词转为小写,复数形式转换为单数,剔除重复等。最后,你得到了一个更小的唯一符号列表,像“*the*”、“*silence*”、“*of*”和“*lamb*”。

一旦提取所有符号,就可以像使用书籍索引中的关键词一样使用它们。不是将电影标题整体添加到索引中,而是独立地添加每个词,并附带一个指向其所在文档的指针。图 8-12 所示为一个简化的倒排索引的结构。

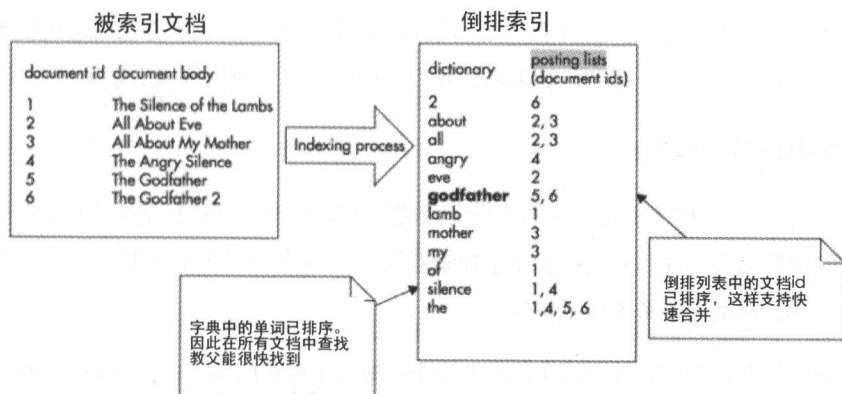


图 8-12 倒排索引的结构

从图 8-12 可以看到，每个符号旁边的 document ID 都已被排序，这样可以在一个表里快速执行搜索，以及列表合并。当要查找包含特定单词的文档时，首先在字典中查找这些词，然后合并它们的倒排表。

提示

倒排索引结构与书籍索引类似，它是单词（符号）的排序列表，每个单词指向一个页码（document ID）的排序列表。

搜索短语（“silence”和“lamb”）需要合并这两个单词的倒排表，并查找其交集。搜索短语（“silence”或“lamb”）需要合并这两个单词的倒排表，并查找其并集。在这两种情况下，合并都很快，因为 document ID 列表是按序存储的。搜索“和”操作代价相对小一点，因为可以跳过很多 document ID，合并结果列表大小也比“或”查询情况少很多。在两种情况下，搜索操作的代价仍然很高，时间复杂度为 $O(n)$ （ n 是倒排表的大小）。

理解了倒排表的工作原理，可以帮助理解为什么在全文搜索中 OR（或）条件代价特别高，为什么搜索引擎需要这么多内存资源。在上百万文档的情况下，每个文档包含几千个单词，一个倒排索引比普通索引增长快很多，因为每个文档中的每个单词都必须索引。

理解了不同类型的索引的工作方式，可以帮助你设计更高效的 NoSQL 数据模型，因为 NoSQL 数据建模设计方式更贴近设计索引，而不是关系数据模式。事实上，Cassandra 最初在 Facebook 用来实现倒排索引，并通过文本框进行搜索。^[w27]之前说过，我不推荐

从零开始实现一个全文本搜索引擎，因为代价非常昂贵。相反，我推荐使用通用搜索引擎作为基础设施的一个额外补充。接下来，我们快速浏览一下通用搜索引擎集成模式。

使用专用搜索引擎

我之前提过，搜索引擎是专用于搜索的数据存储。它们尤其擅长全文本搜索，但也允许索引其他数据类型并高效地执行复杂搜索查询。当需要基于大数据集构建一个复杂搜索功能的时候，你就需要考虑搜索引擎。

要展示复杂搜索的特性，可以看看之前提到的汽车目录网站。一些网站同时有十几万辆汽车销售，必须提供更高级的查询条件（否则用户会收到大量不相关的查询结果）。因此，你会见到高级搜索表单提供十多个查询字段框，你可以搜索任何条件，包括任意文本、商标、模式、最低最高价、最低最高里程数、燃油类型、传动类型、马力、电子后视镜和加热座椅等饰品的颜色。更复杂的情况是，执行搜索后，你可以把数据的一部分展示给用户，好让他们提供额外过滤条件来进一步缩小查询范围，而不用重新搜索。

像这样的复杂搜索功能是专用搜索引擎真正的亮点所在。你不用在应用中自己实现复杂的方案，可以直接使用现有的搜索引擎。有一些很流行的搜索引擎，比如 Amazon CloudSearch 和 Azure Search，以及一些开源产品比如 Elasticsearch、Solr 和 Sphinx。

如果使用托管服务，你不需要自己操作、伸缩及管理这些组件。搜索引擎，尤其是一些前沿技术，在生产环境中很难伸缩和操作，除非你有经验丰富的工程师熟悉这些技术。你可能会牺牲一些灵活性，一些边界特性，但这可以降低你的产品上市的时间及运维复杂度。

深入探讨如何对搜索引擎进行配置、伸缩和操作超出了本书的范围，不过我们可以快速地看看如何与其集成。例如，如果你决定在之前的汽车销售网站上使用 Elasticsearch 作为搜索引擎，你需要将其部署在数据中心上，并对所有汽车建立索引。使用 Elasticsearch 索引汽车非常简单，因为它不需要任何预定义模式。你只需要简单地为每辆车生成 JSON 文档并提交给 Elasticsearch，它会自动建立索引。此外，要保持搜索索引数据与主数据存储同步，需要在汽车元数据变化时及时更新 Elasticsearch 中的索引。

提示

在搜索引擎中索引数据的一个常用方式是使用作业队列（尤其是当搜索引擎更新准实时化的时候）。每当有人修改汽车元数据时，系统会发布这辆汽车的异步消息到队列。稍后，队列工作者从队列中获取消息，构建包含所有信息的 JSON 文档，并提交给搜索引擎覆盖之前的数据。

图 8-13 所示为搜索引擎部署示意图。所有的搜索操作都由搜索引擎执行，然后搜索结果会通过主数据存储中的实时数据进行丰富（如果有必要的话）。另一方面，用户修改汽车操作会直接写到主数据存储，并通过队列和异步索引过程更新搜索索引。

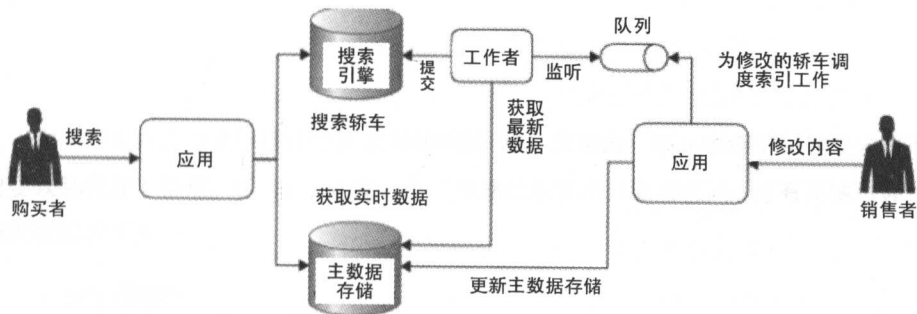


图 8-13 倒排索引

将所有汽车数据索引在 Elasticsearch 中，就可以执行复杂查询，例如“获取丰田所有提到‘畅销’的、在 2000 年到 2005 年生产的、带电子车窗、标记为特价优惠的汽车，并按价格和属性如地点、模型、颜色等排序”。

搜索引擎在 NoSQL Web 栈工具集中是一个重要的工具，强烈推荐大家熟悉至少两个平台并能够有效地选择和使用搜索引擎。

小结

高效地进行数据搜索是一个重要的挑战。本章提到的关键概念包括采用针对特定访问模式进行优化的存储方式，以及一个搜索伸缩的主要工具——索引。

还要记住 NoSQL 引入了一种新的思考数据的方式，在这种方式下，需要识别数据用例和访问模式，并抽取数据模型，而不是采用传统方式设计数据结构。此外，可以使用专用搜索引擎完善基础架构，处理复杂的搜索问题。

我们深入地探索了搜索、索引和数据建模技术^[16, 19, w27-w28, w47-w48, W71]。数据搜索是本书纯技术章节中的最后一部分；接下来，我们看看 Web 初创公司面临的伸缩性问题的其他维度。

9

伸缩性的其他维度

本书主要内容是关于设计与开发可伸缩的 Web 应用的一些技术细节，这些内容的主要受众是软件工程师。事实上，开发一个可伸缩的系统不只是写代码，还有其他可伸缩的维度也需要关注。

- **运维伸缩性**

生产环境能够运行多少台服务器？系统一旦部署在上百台服务器上，如何高效地管理这些服务器就会成为一个挑战。如果每增加 20 台服务器就需要多招一个运维管理员，那么运维就没法快速廉价地伸缩。

- **个人影响力的伸缩性**

你个人能为客户和公司创造多大的价值？随着创业公司的成长，你个人的影响力也应该同步增长。通过扩展你的职责范围及对各个业务主管施加影响，你的工作效率和个人绩效会变得更高。

- **工程师团队的伸缩性**

创业公司的工程师人数达到多少后工作效率会下降？随着公司的成长，公司需要在不影响工作效率的情况下招募更多的工程师壮大技术部门。这就意味着需要建立良好的工程师文化，规划合理的团队和系统架构，保证大家能够可伸缩地并行开发与协作。

伴随着公司的成长与个人的进步,你应该能深刻体会这些非技术的伸缩性。为了应用能够真正可伸缩,你需要使技术团队规模可伸缩,支持的服务器数量可伸缩,你自己的个人影响力和生产力可伸缩,同时还要使成本与代价(时间与金钱)最小化。本章我们会探讨一些帮助创业公司可伸缩的方式。

自动化实现生产力可伸缩

“如果你想要某件事情发生,命令它。如果你想要某件事情重复发生,自动化它。”

——Ivan Kirigin

创业的一个重要哲学是用一个比较低的成本增长率获得公司价值的持续扩展。意思是,获取第二个一百万用户要比获取第一个一百万用户付出的成本更低。虽然看起来有点违反直觉,但是互联网创业的单位成本确实是随着时间的推移下降的。这也就意味着你的公司随着逐渐成长必须变得更高效。每个用户的成本,每个交易的成本,每次检验的成本,无论你以何种方式计算,都应该随着规模的增大而降低。

现代的技术创业公司可以达到一个不可思议的用户员工比,一个很小的技术团队开发的产品可以供上千万甚至上亿用户使用。这种效率约束使得伸缩性更具有挑战性,你设计与开发的系统必须随着其规模增大而更高效。要想达到这样的效率,需要尽可能地实现自动化。我们看一下有哪些可以实现自动化进而改善效率的常用方法。

测试

要构建一个可伸缩的 Web 应用,测试是第一件需要实现自动化的事情。尽管业务人员和工程师们花了大概十几年的时间才接受自动化测试,但是现在自动化测试已经成为软件行业的一个事实上的标准。使用自动化测试的一个主要原因在于手工测试的成本增加远远高于自动化测试的成本。

图 9-1 所示为手工测试和自动化测试的总体成本。如果只依靠手工测试,是不需要任何前置成本的。雇个测试工程师,测试工程师在每次发布前进行应用测试。开始的时候,这种成本是非常小的,但是增长却很快。每次开发新功能都需要对新功能进行测试,而前

面已经发布的各种功能也需要进行测试,确保开发新功能不会破坏现有的产品功能。也就是说,每次发布的测试成本都比前一次发布的测试成本更高(测试效率随时间推移逐步降低)。这也会使软件的版本的发布周期逐渐变长,因为随着软件规模的增长,测试需要越来越长的时间。

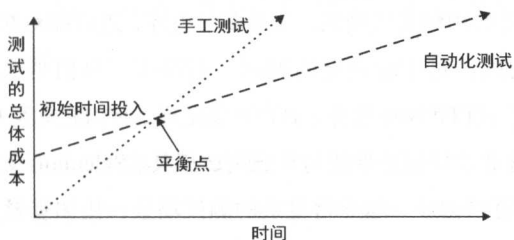


图 9-1 手工测试和自动化测试的总体成本

自动化测试需要一些前置投资,你需要安装自动化测试工具,部署持续集成服务器,但是后续投入却是相对便宜的。一旦完成初始安装,只需要为新功能创建测试就可以了,测试已经存在的功能几乎不需要花费任何成本。随着时间的推移,测试会变得越来越高效,每一次发布测试,已测试的代码和待测试的代码之比都在增加。最终,会到达一个平衡点,然后自动化测试的总体成本会低于手工测试的成本。

虽然自动化测试的成本和收益很难度量,但是可以预期一家创业公司经过 3~6 个月的开发后可以达到平衡点。自动化测试的收益依赖于具体的业务场景,比如,核心业务的数目、团队的规模、缺陷容忍度。

此外对于时间和金钱的投入,自动化测试也会为团队带来信心,团队可以面对变化快速做出反应,以更激进的方式对系统进行重构,这些优点对于创业公司而言特别重要。通过自动化测试和持续集成,可以加速开发周期,而不必在系统发布前花费太多的时间去测试。通过自动化测试工具,可以在系统早期就进行集成,更频繁地发布,更好地对市场需求做出响应。

提示

通过自动化测试构建起自信的基础,你的团队会变成 Robert C. Martin 所说的那种无畏工程师。无畏工程师不会害怕任何变更,他们完全掌控他们的软件,他们自信 BUG 在产品发布之前一定能被发现。

根据你具体使用的开发语言和技术，可以使用不同的工具实现自动化测试。首先，你最需要的自动化测试是单元测试，单元测试是不需要部署其他组件就能执行的一类测试。单元测试执行非常快，维护非常容易，几乎不会带来负面影响，而且在代码变更时容易进行 Debug 和错误修复。

此外，你可能还需要用到集成测试，涉及多个组件，进行端对端的测试，通过公开接口进行整个应用的测试。对于端对端的测试，有很多工具值得推荐，比如 Jmeter 及 Selenium。Jmeter 对于 HTTP Web 服务、HTTP 重定向、HTTP 头及 Cookie 测试都非常有用。同时，Jmeter 也是非常好用的性能与负载测试工具。Selenium 支持在测试内远程控制一个浏览器。所以，可以创建一组非常复杂的测试场景，比如登录、购买、订阅。使用 Selenium，可以将手工测试能做的任何事情都自动化并放入你的自动化测试套件中。

一旦实现测试自动化，下一步的自动化构建、部署与发布过程就会有一个坚实的基础。下面我们看一下如何在这些领域实现自动化。

构建与部署

提高效率的第二步就是自动化整个构建、测试和部署过程。跟手工测试一样，手工部署也是一个坑。随着服务器和服务数量的增多，需要参与的人员也越来越多，服务和服务之间的依赖关系也越复杂，部署的难度也越大。软件发布的困难也因此增大，测试与集成过程越长，发布周期也越长，导致每次发布的功能更庞大更复杂。图 9-2 所示为手工发布导致的恶性循环与自动发布产生的良性循环。

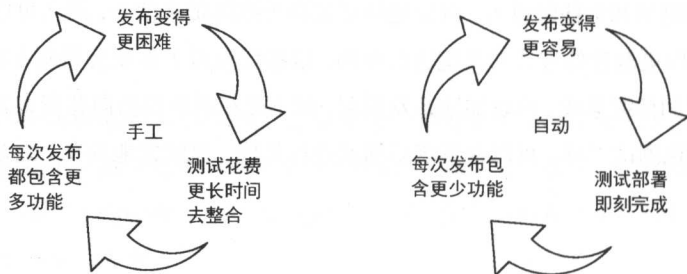


图 9-2 手工部署与自动化部署

打破手工发布恶性循环的最佳实践就是通过引入持续集成、持续交付与持续部署，实现构建、测试、部署的过程自动化。

持续集成是实现自动化演进的第一步。允许工程师随时向公共分支提交代码，并立即进行自动化测试。编写自动化测试及提交代码到公共分支，工程师可以及早发现集成错误。这样可以节省代码合并的时间及减少集成发布带来的麻烦。有一个测试通过了的稳定的集成分支（每次代码提交都会在集成服务器上执行一次自动化测试），就可以在任何时间将代码部署到生产环境中。持续集成本身并不会进行部署，他只是确保每次代码提交都能够通过打包构建及自动化测试。

持续交付是自动化演进的第二步。除了跑单元测试及软件打包，持续交付机制会将软件部署到各种测试环境中（这些测试环境通常包括开发、测试、演示三种）。这个过程的一个最主要特点是软件会在无须人工干预的情况下被编译、组装、部署到一个类生产环境中。这就意味着，任何时候当工程师提交了一次代码，自动构建就会被触发，随后软件被发布到开发、测试或者演示环境中，接着自动执行端对端的测试，以验证整个系统是否能正确运行。最终，代码是否可以部署到生产环境，会变成一个商业决策而不只是一个工程师的技术决策。一旦某个变更通过了持续交付的自动化流程，就已经准备好可以发布到生产环境了。在这个阶段，部署到生产环境使用的自动化脚本和部署到演示环境的脚本基本一样，只需要单击一下按钮，或者输入一条命令就可以，无须关注基础设施的复杂性或者服务器的数量。

持续部署是部署流程演进的最后一步，在这一步，代码在没有人工干预的情况下被测试、构建、部署并推送到生产环境。也就是说，每次代码提交到公共分支，都会在没有人工参与的情况下被部署到生产环境服务器。部署非常容易，一天可以部署好几次，而不是几个星期才部署一次。

图 9-3 所示为一个持续部署流程，展示了哪些地方可以通过持续集成、交付、部署实现自动化。理想情况下，软件可以在整个流程中自动化地推进，代码提交到集成分支会触发执行测试并编译打包。然后代码会部署到开发与测试环境，接着在测试环境执行端对端的测试。如果所有这些步骤都成功通过，代码会立即部署到生产服务器，结束持续部署流程。

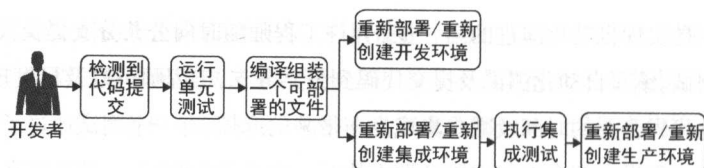


图 9-3 持续部署流程示例

我这些年在很多公司工作过,见证了各种各样的部署和各种各样的自动化水平。例如,我曾经工作过的某个团队,请了三个人每周花 4 个小时去发布软件,这样每年花费大概 300 小时,相当于有个工程师有两个月的时间什么也不干,光是发布软件。当然也有另一种极端,我曾见过有的团队每次提交代码都会将软件部署在生产环境,从提交代码到完成上线只用 15 分钟。

提示

如果你想实现可伸缩性,测试与部署必须实现自动化。如果你完成一次构建、测试、部署上线需要点一次以上的按钮,你就需要更加自动化。

协助完成持续部署的工具和平台最近几年越来越多,但是安装一个全栈的自动部署系统依然是一件相当复杂的事情。安装一个自动部署系统之所以有挑战是因为这件事需要拥有完整的 DevOps 技能。你需要编码测试的开发技能 (Dev), 还要有安装服务器、部署系统、管理配置的运维经验 (Ops)。

你最好祈祷团队里有一个有经验的 DevOps 工程师帮你完成系统安装,如果你需要自己安装持续部署系统,我建议你使用 Jenkins (一个开源产品), 或者 Atlassian Bamboo (一个商业产品, 但是功能更强大) 作为系统控制工具。

此外,为了配置持续部署工具 (比如 Jenkins), 你还需要决定如何管理服务器的配置。目的是在源控制系统中存储服务器配置,以便在任何时间点重建服务器并跟踪变化。如果想在管理服务器方面获得可伸缩性的能力,那么使用专门的工具比如 Chef 或者 Puppet 管理配置绝对是非常必要的,这样你可以完成服务器定义,并随时创建这些服务器的实例。

提示

手工控制超过十几台服务器,或者允许不同服务器之间使用不同配置,这绝

对是一个灾难。每种服务器类型（比如数据库服务器或者 Web 服务器）都应该用配置管理工具创建一个服务器定义。这个定义用于创建特定服务器类型的实例，确保所有服务器实例都是一样的。遵循这个过程，可以任意伸缩服务器数量而不会增加管理成本的开销。不论你要部署一台服务器还是部署一百台服务器，只要完成一次定义，部署多少台服务器都不是难事。

完成服务器定义及持续部署工具配置，你就可以准备服务器的自动化部署了。要实现这个目标，还需要编写一些客户脚本，知道哪些服务器以何种顺序进行部署。还可能需要进行一些额外的命令，比如清缓存，重启 Web 服务器，将服务器从负载均衡器中剔除。

为了部署到生产环境，持续部署系统（以及客户脚本）可能需要和云主机提供商集成，创建服务器镜像（比如亚马逊机器镜像 AMI），构建新服务器实例，从负载均衡器中移除或者添加服务器。实现这些功能并不是只有一种办法，根据你的基础设施情况、技能水平、主观意愿，你可以选择不同的方案。比如，如果你使用亚马逊云主机，就可以用 AMI 进行部署实现自动伸缩及自动服务器替换。在这个例子中，你的持续部署系统看起来可能是这样的：

1. 开发人员提交代码到主干分支。
2. GitHub 向持续部署工具发送通知信息。
3. 系统检查代码；执行单元测试；构建打包集合；测试结果、文档、其他资料被压缩并推送到持久存储中（比如亚马逊 S3）。
4. 上述过程成功后，使用亚马逊 API 创建一个新的服务器实例以构建一个新的服务器镜像。这个实例是用最新的生产环境 AMI 镜像恢复出来的。
5. 使用配置管理工具对这个实例进行升级，将程序包升级到最新，部署应用的最新版本。通过这种方式，只需执行一次安装、配置、依赖打包。
6. 实例就绪后，将其快照作为最新的 AMI 镜像，供后面创建新的服务器实例使用。
7. 将新创建的服务器镜像部署到测试、验收集群，然后使用端对端测试工具，比如 Selenium 或者 Jmeter 验证其正确性。

8. 将最新创建的 AMI 镜像标记为生产环境就绪，进入生产环境部署环节。

9. 重新部署生产环境 Web 服务器集群，一种方式是：更新负载均衡器，将 Web 服务器一台接一台地顺序移除、更新，然后添加回负载均衡器；另一种方式是先将 Web 服务器集群扩大一倍，新增加的服务器都是最新的代码版本，然后将旧代码的服务器关闭。

拥有这样一套持续部署系统可以让你非常快速地交付软件。可以一天部署好几次新功能，可以快速进行 A/B 测试以验证方案的有效性，而不必等上好几个星期才能得到用户的反馈。

唯一的问题是“如何才能保证每次提交都能通过所有步骤直到生产环境，而不会因为各种原因中断”，有一些持续部署的最佳实践可供参考。

- 给所有代码写单元测试，单元测试覆盖率至少达到 85% 才能保证一个较高的稳定性。
- 为所有关键路径创建端对端的测试用例，比如注册、购买、添加商品到购物车、登录、订阅消息等。使用诸如 Selenium 这样的工具，可以在代码部署到生产环境之前就验证系统的主要功能是否正确。
- 使用功能开关即时开启或者关闭某些功能。处于隐藏模式的功能对一般用户不可见，处于不可用状态下的功能对所有用户不可见。使用功能开关，可以快速关闭那些有故障的功能而无须重新部署启动服务器。
- 使用 A/B 测试及功能开关可以在一个较小的用户集上测试某个新功能。将某些功能设为隐藏模式，仅对部分用户可见（大概 2% 的用户），可以以一种较低的风险进行测试。在 A/B 测试期间，还能收集一些商业数据，看看这些功能是否被用户使用，是否提升了商业指标（比如，提升了交易量，增加用户黏度）。
- 大范围使用监控工具，采集所有应用的数据，对重要数据进行监控和报警，第一时间发现哪里出了故障。

遵循这些最佳实践，可以帮你实现持续部署能力，快速将更新部署到生产环境，同时降低故障的概率。不管你对代码如何测试，你还是会遇到一些故障，需要快速高效地处理。为了伸缩你的运维能力，自动化的监控和报警是绝对必要的，下面我们看下这方面的内容。

监控与报警

自动化监控报警的主要目的是通过降低失效恢复时间 (MTTR) 提高系统可用性。当你的生产环境只有两台服务器时, 搞自动化监控、失效检测、报警似乎有点太奢侈, 但是随着服务器数目的增长, 自动化对于提高运维效率绝对有必要。失效恢复时间由 4 个部分组成。

失效恢复时间 (MTTR) = 发现时间 + 响应时间 + 调查时间 + 修复时间

发现时间是指业务上意识到系统出现问题需要的时间。在小公司, 失效通常是用户报告的, 或者是员工注意到了问题。所以就会出现这样的情况: 系统已经出问题好几个小时甚至好几天了, 才有人报告问题。结果就是极糟的用户体验和极差的可用性指标。使用自动监控, 可以将故障发现时间缩短至几分钟内。

第二个部分是响应时间。同样, 对于小公司, 可能需要几个小时才能找到正确的人去处理问题。先是不知道该去找谁, 找到了又不知道他的联系方式, 好不容易联系上了, 他可能没带计算机在身边, 或者是没有密码登录不到出问题的服务器上, 甚至有可能正在描述问题但他的手机突然没电或者欠费了。最终就是故障响应时间完全不可预估, 有可能要花上几个小时才能着手查找问题。随着公司的成长, 运维团队需要实现自动化故障通知, 线上问题能够快速分发给正确的人, 他可以在几分钟之内做好准备去处理问题。此外, 还需要制定一个规范, 约定谁, 在什么时候, 如何处理何种类型的故障。通过一个清晰的规范结合自动化报警, 可以将响应时间从几个小时降低到十几分钟。

MTTR 最后可以控制的时间是调查时间和修复时间, 这两部分时间独立于监控和报警。在小公司, 如果系统出了故障, 工程师会登录到生产环境服务器, 打开日志文件, 从最后看起, 以定位故障原因。很多时候, 是数据存储与外部系统失效引起报警并产生复杂的连锁反应, 工程师需要查看十几台服务器上的日志文件寻找引发故障的根本原因, 这是一个非常耗时的过程。

如果想加速 Debug 过程及对待处理的问题获得一个完整的印象, 需要收集关键数据指标及对日志进行聚合。通过监控系统内部, 可以快速定位究竟是哪个组件响应变慢或者出了故障。也可以通过相关联的多个数据指标推测可能的连锁反应。最后, 通过聚合日志信息, 可以快速搜索日志信息找到故障点, 减少调查时间。

收集各种类型的数据指标除了可以降低 MTTR，还有助于深入了解业务内幕发现业务发展趋势。如果想从监控中获得尽可能多的信息，建议你收集以下 4 种不同类型的数据指标。

- **操作系统数据指标**可以让你看到硬件、网络、操作系统的状态。在这个层面上，可以收集的信息包括 CPU 负载、内存使用、正在运行的进程数目、网络连接、磁盘 I/O。这些指标主要供系统管理员及 DevOps 人员评估系统能力及性能调优使用。
- **一般服务器数据指标**指那些可以通过 Web 服务器、应用容器、数据库、消息队列、缓存服务器获取的数据指标。在这个层面，你可以收集如下数据指标：每秒数据库事务数，等待锁释放花费的时间，每秒 Web 请求数目，最长队列中的消息数目，缓存服务器的命中率。这些数据指标有助于你更深入地了解组件的内部信息进而发现其瓶颈。
- **应用数据指标**是应用程序产生的供分析用的数据指标，这些数据深入到应用程序内部，可以了解应用程序内部的状况。应用程序级的数据指标的一个例子是对外部系统的调用，比如数据库、对象缓存、第三方服务、数据存储等。每一种外部调用，你都想知道调用出错率，调用次数，调用花费的时间。通过分析这些指标，可以快速发现系统的瓶颈在哪里，哪一个服务调用比较慢，处理能力的变化趋势是什么。在这个层面，你可能收集的指标有：每个 Web 服务完成一次响应需要花费的时间有多长，不同的功能被使用的次数是什么样的。这些指标的主要意图是让工程师了解代码如何运行，面对的访问压力主要是什么。
- **业务数据指标**追踪业务事件。比如，追踪资金流向，用户账号创建，加入到购物车的商品数量，每分钟登录系统的用户数量。这些指标可以让工程师在几秒钟之内就能快速判断是否出现了一个影响用户行为的故障。这些指标还可以将一个技术问题转换成其对应的业务影响，比如资金丢失或者用户登录失败。知道这些问题的业务影响，可以通过观察用户活动更高效地进行系统升级及失效验证。

如果没有这些数据指标，进行 Web 应用伸缩就像是瞎子开车。事实上，我建议部署到生产环境的每一个服务都能加上数据指标监控，这样才能有针对性地诊断并进行伸缩。现在，我们看一下在实践中如何进行监控。

监控报警通常的实现方案是在每台服务器上安装监控代理。监控代理依据配置情况收集这台服务器及其上部署的所有服务的数据指标。根据服务器角色的不同,代理作为一个插件可以收集各种来源的数据指标,比如数据库进程、消息队列、应用服务器,以及缓存。每个监控代理通常都会聚合几十、几百,甚至几千的数据指标。代理会将这些数据指标周期性地(比如每分钟或者每五分钟)发送给中央监控服务,中央监控服务通常是一个云服务或者是一个内部部署的监控服务器。

数据指标被发送到监控服务后,这些数据首先会被记录下来,以便控制面板显示和绘图。同时监控服务检查这些指标的值和变化率,如果超过了设定的安全阈值,就会发送报警信息。比如,如果数据库连接数达到了某个值(一旦超过这个值,数据库查询会变慢,导致连接数进一步上升)就会发送一条短信给工程师。

将数据指标从各个服务器推送到中央监控服务,就可以按服务器或者服务器集群进行可视化展示,也可以按服务器或者服务器集群设置报警。比如,你可以监控每台服务器的空闲磁盘空间,因为有的服务器比其他服务器用磁盘用得更多。另一方面,监控数据库与从服务器连接数的时候,你可能对整个集群的总体连接数更感兴趣,想要看到整个系统的全貌,无论机器是不是被访问,是不是正在被维护。

除了通过内部监控代理及监控服务获取数据指标,还可以选择外部服务等级协议(SLA)监控服务。使用第三方SLA监控服务的好处是它可以像用户一样通过外部网络连接你的服务。这样,就可以探测到网络故障,路由/虚拟专用网络(VPN)配置问题,域名服务(DNS)问题,负载均衡配置问题,安全套接层(SSL)认证过期,以及其他从网络内部不能探测到的问题。此外,某些SLA监控服务可以从世界各地使用不同的设备(各种移动网络及低带宽的连接)对系统进行性能测试。最终,它们可以为你提供海量的有价值的数 据,让你优化你的用户体验,实现各种故障类型的报警。

图 9-4 所示为监控与报警的实现机制。监控代理部署在各个服务器上,数据指标被推送到中央监控服务,报警规则也配置在监控服务内。此外,还可以使用外部SLA监控服务从用户视角监控系统性能与可用性。

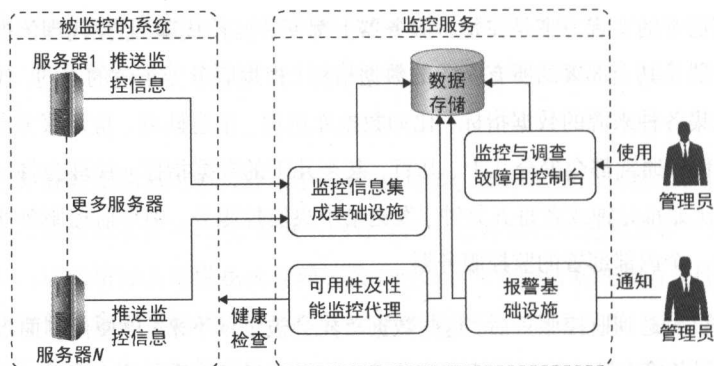


图 9-4 监控与报警配置示例

可以想象，部署这样复杂的一个监控系统会涉及很多组件。针对需要监控的服务类型安装监控代理及各种插件。还需要能发布任意数据指标，不管是应用程序代码还是各种业务指标。还要配置这些代理推送数据到聚合服务，在服务端需要配置图表、阈值、报警、控制台。所有这些组件必须有效组合起来，而这些组件本身也需要具有可伸缩性，这又需要一大堆工作。所以这就是我为什么推荐使用第三方的、基于云的监控服务，而不是部署自己的解决方案。

迄今为止（2014 年），我见过最好的监控服务是 Datadog，我强烈推荐在选择一个解决方案之前先进行测试。Datadog 的优点是功能丰富，易于集成，易于使用。Datadog 提供了各种插件用以监控不同的开源服务器，无须编写任何代码就可以监控数据存储、消息队列、Web 服务器等。此外，它还提供了一个简单的应用程序接口（API）与客户程序包，你可以在应用程序代码内部发送各种数据指标。最后，它还有一个很友好的用户接口（UI），你可以花很少的工夫配置图表、控制台阈值，以及报警。

监控领域的竞争最近几年愈演愈烈，有一些非常不错的产品出现，例如 Stackdriver、New Relic、Server Density 等，可以关注一下。有些服务提供商，比如 Server Density，提供外部 SLA 监控作为整体方案的一部分，有些则不会。如果你想使用独立的外部 SAL 监控提供商，推荐考虑 Pingdom、Moniris，以及 Keynote。

通过监控、报警，以及快速响应流程，应该可以将 MTTR 降到十几分钟。如果你想进一步降低调查与 Debug 的时间，可能还需要实现一个日志聚合与索引方案。

日志聚合

当你的应用比较小的时候,只有几台服务器,那么日志聚合并不是特别必要。问题通常总是出现在某台服务器上,你只要打开某几个日志文件,使用 `tail` 或者 `grep` 命令寻找问题原因即可。但是,当系统规模和服务器数量增长到几十台(然后就是几百台)的时候,你就没办法手工搜索日志了。甚至在只有十台服务器的情况下,`tail` 和 `grep` 也不好使,相关的日志同时分布在所有服务器的所有日志文件里。前端层的请求也许会级联调用十几个 Web 服务,搜索所有这些机器上的所有日志文件就变成一个非常艰巨的挑战。要想高效地搜索这些日志快速 Debug,需要某种方法将所有日志集中到一起。目前有几种常用的方法实现这一点。

一个基本的方案,你可以将日志直接记录到数据存储系统,而不是记录到本地日志文件。这种方案好的一面是不需要移动日志文件就可以将日志聚合在一起。差的一面是所有的组件都要依赖数据存储系统的性能和可用性。由于这是一种不必要的耦合,所以将日志写入数据存储系统这个方案并不被推荐。

更好的方案是将日志写到本地文件,然后将这些日志传送给中央日志服务。最简单的方式是在每台服务器上安装一个日志转发代理,然后以流的方式将日志转发给中央日志服务器。这个方案的最大好处是简单,只需要一个日志服务器及在每台服务器上安装一个日志转发代理即可。

有很多开源产品实现了以流的方式将日志发送给中央日志服务器,这类产品的一个典型例子就是 `Fluentd`。`Fluentd` 使用简单,可用性高,伸缩性好,功能丰富。日志转发代理可以遍历多个日志文件,执行复杂的过滤与转换操作,然后将日志转发给中央日志服务器。除了将分散的日志文件放在一台服务器上,还要合并不同数据源的事件,并统一时间格式,不然光是处理日志中不同时区和不同格式的时间就让人崩溃。

将日志以流的方式发送到中央服务器是一种进步,但是如果日志以非结构化的方式存储,那么还要花一些功夫才能使其变得容易检索。日志越多,执行检索的速度越慢。看你的需要,也许你可以更进一步使用日志索引加速搜索,以及使日志数据在整个公司更容易使用。

图 9-5 所示为一个完整的日志聚合部署模型。每台服务器上都安装一个日志转发代

理, 对其进行配置决定哪些日志要转发, 如何过滤与格式转换, 日志转发到哪里。然后日志以数据流的方式发送到一组搜索引擎服务器, 在那里数据被持久存储并构建索引, 以便更高效地进行日志检索。此外, 日志处理平台还会提供一个基于 Web 的访问接口, 可以更容易地进行日志搜索。

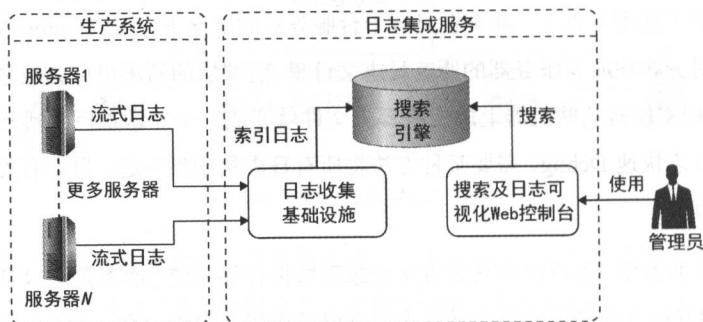


图 9-5 日志集成工作流示例

部署与管理一堆组件进行日志聚合构建索引又是一大堆工作。本书写作期间, 市面上最知名、最引人注意的解决方案是 Splunk。不幸的是, Splunk 非常昂贵, 不是所有创业公司都能从其提供的价值中收回成本。有些云供应商也会有一些基本的日志聚合服务, 比如亚马逊 CloudWatch Logs 或者 Azure Diagnostics, 但是这些服务缺乏足够的弹性。你也可以考虑那些独立的日志处理服务, 比如 Loggy, 不论你的主机平台是什么, Loggy 都提供很好的功能服务。

如果你不考虑将应用日志分发给第三方或者部署第三方的代理到你的服务器上, 你也可以考虑自己部署一个开源的解决方案。这种情况下, 我推荐 Logstash, 一个功能丰富、可伸缩的日志索引平台。Logstash 使用自己的日志转发代理将日志转发给 Elasticsearch 搜索引擎。它还集成了一个叫做 Kibana 的 Web 接口, 你可以准实时地进行文本检索及实现日志可视化。使用 Logstash 唯一的缺点是要自己学习如何配置并使用它。使用 Logstash 并不简单, 即使是一个简单的部署, 也需要大量的配置步骤,^[18~19] 如果没有一个有经验的系统管理员, 可能会是一个很重的负担, 超过你能获得的收益。

自动化测试、配置管理、部署、监控、报警, 以及将日志统一收集到中央服务器, 这些工作可以让你以一种可伸缩的方式运维你的 Web 应用。接下来, 让我们看看随着创业公司的成长, 你如何做到你个人生产能力的可伸缩性。

个人可伸缩

创业公司最有吸引力的地方就在于其指数级增长的可能性。快速高效地增长让投资人和创业者都能获益。为了实现这种指数级增长，你也需要具有快速的成长性。你需要变得更高效，能为客户为公司创造更多价值。下面我们看看在一家创业公司工作会面临的主要挑战，如何实现个人成长，使工作绩效和工作乐趣都最大化。

加班不是一种伸缩性方案

在巨大的压力、有限的资源、压缩的工期、严重的不确定下工作会让人神经紧张，而这正是创业公司工作的常态。创业公司是一杯让人情绪爆发的鸡尾酒，引人入胜又让人筋疲力尽，同时又收获满满，不过也要小心不要让创业变成一场无目标的比赛，让工作变成一种无思考的冲动。

工作时间越长工作成果越多似乎是一件自然的事。给自己多一点压力，每天都工作很长时间，甚至周末都在工作，你精力充沛、目标明确、充满希望、相信未来，看起来这是正确的做事方式，似乎做出的牺牲也并不大，而且那种被需要的感觉也很赞，感觉自己就是一个英雄。

问题是这不是一个长久之计，加班工作是实现个人生产力伸缩性的一种非常糟糕的办法。超过正常工作时间的加班时间越多，精神状况就越差，创造力越低，注意力越分散，眼界和决策力都会退化。另外，你还会渐渐变得多疑、暴躁、易怒。你会讨厌那些工作时间比你少的人，面对越来越多的工作你会变得无助和沮丧，甚至会开始厌恶那些你曾经挚爱与渴望的事业，只有一种方式可以压抑这种焦虑，那就是更疯狂的加班。这是一种病，被称为过劳症。

过劳是创业公司工作的大敌，它会偷偷地占据你，使你变得盲目，最后被它控制。就像一个死循环，你越努力工作，就会越累，就越不能找到高效的工作方法，你就越需要更努力的工作。每个人都或多或少有过过劳的体验，就我自己而言，真是一种糟糕透了的感觉，花了好几个月的时间才完全恢复过来。我的体会是，3~9个月的高强度工作（在极大的压力下每周工作45到60小时）就可以体验到相当程度的过劳了。

图9-6所示为加班对劳动产出的影响。开始的时候，劳动产出会随着加班时间增加，

但是用不了多长时间，加班的劳动产出会逐渐下降，然后随着加班情况的持续，劳动产出会变得和正常工作一样。再往后持续加班只会导致更少的劳动产出，直到最后受不了以闪人告终。

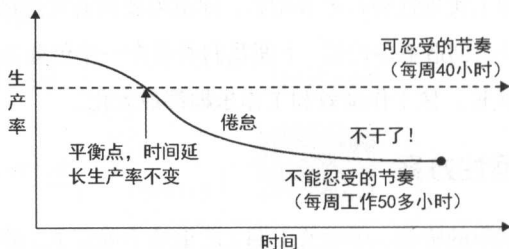


图 9-6 加班的生产率

提示

如果你在一家创业公司工作，那么很有可能你已经或正在体验过劳症的感觉。坏消息是没有一种快速简单的方法可以从这种症状中恢复。少一点工作，多一点锻炼，多花一点时间陪朋友和家人，即使这样，要完全恢复还是要花几个月的时间。还有一种较快速的方法是休个长假（3 到 6 个星期），或者干脆离开这个项目换一个轻松一点的工作。在你的职业生涯中一定会经历几次这样的过劳症，然后才会学会如何在这些症状的早期就发现它并通过自我管理的方式避免。

不要让自己陷入超级能干然后不久崩溃的循环之中，以一种更有效更健康的方式将努力程度维持在一个可持续的水平上。每个人的情况都不相同，动力不同，干劲不同，个性不同，职位不同，我个人的经验是，每周工作超过 50 小时就很危险，每周工作超过 60 小时大概 1 个月左右就会出现过劳症。

你的时间是你最珍贵、最难以转换的财富之一。你的一个小时就是恒定的 60 分钟，没有办法对时间进行伸缩。与其试着工作更长时间，不如寻找一些方法在每周的 40 个小时里为客户为公司为同事创造更多的价值。这句话听起来有点像一句空洞的口号，不过真的要学习一下如何更聪明地工作，而不是更努力地工作。

自我管理

使自己生产力最大化的方式是将自己当作一个项目来对待,自己的任务就是项目的任务。项目管理的时候,有三个杠杆对项目进行平衡:范围、成本和时间。

任何时候,增加或者减少范围、成本或者时间中的一个,其余两个变量都必须调整以达到新的平衡。如果你增加工作量,就必须投入更多资源或者延长项目时间。如果你想减少时间以提前交付项目,就必须缩减工作范围或者增加资源。如果减少了投入的资源,就必须砍需求或者延期。

图 9-7 所示为项目管理三角,并且加了一些额外的说明以帮助记忆这些应对策略。首先,需要接受的事实是项目管理是一个关于权衡的艺术,花更多的时间和金钱却做更少的东西。当你用这种方式思考工作的时候,就会发现一些其他的办法去平衡项目而不是加班加点。

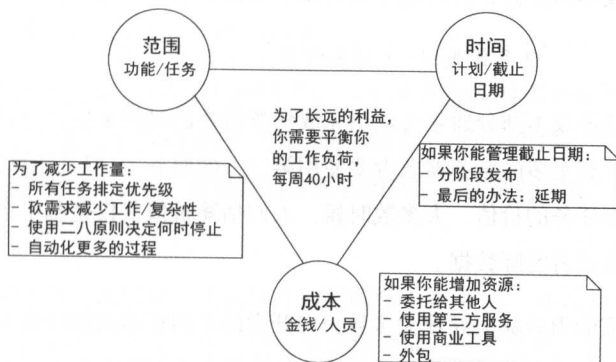


图 9-7 项目管理级别

提示

管理三角里必须考虑质量这个因素。我建议构建自动化测试,确保高质量的代码是功能范围的一部分。质量不是可以权衡考虑的方面,如果你想保持一个长久的高效率,就不要试图通过降低质量缩减范围。牺牲代码质量就像是跟 Tony Soprano (《黑道家族》的主角) 借钱。当你工期仓促压力巨大的时候,牺牲质量似乎是个好主意,但是或迟或早,你都要偿还这笔高利贷(周末加班就是贷款的利息)。

我们进一步看看如何影响工作量、成本及工期，从而使工作负荷更具可持续性。

影响工作范围

“没有数据支持，观点不足采信。”

——W.Edwards Deming

工作范围通常是最容易平衡工作负载的方式，也是最有调整空间的地方。要更好地管理工作负载的第一步就是要意识到，当你有一个新的任务，或者现有任务需要增加内容的时候，你需要重新评估完成时间，增加可用的资源或者砍掉一些需求范围。

做任何事情都要花费时间，因此再分配给其他任务的时间就会变少。这似乎是显而易见的，但是学习如何基于成本和价值区分任务优先级是管理工作范围最重要的技能。毕竟，一个创业公司想要生存下去最重要的事情就是做正确的事。要想高效地区分任务优先级，需要知道它们的成本和价值，然后基于相对成本价值率划分优先级。

$$\text{任务优先级} = (\text{任务价值}) / (\text{任务成本})$$

等式里面的任务成本部分通常可以用完成任务需要的时间来估计。虽然大一点的项目包含了财务成本（买更多的服务器或者要使用第三方的服务），但是估算起来并不困难，真正困难的是估算任务的价值。大多数时候，人们估算价值仅仅是基于个人的直觉感受，而不是过往的经验或者实际数据。

这种评估价值能力的缺失导致大多数公司开发的根本就没什么人用。我见证过人们仅仅根据某个人的直觉感受产生的愿景，没有任何数据方面的验证，就拼命做了很多没有意义的事。事实上，根据一个没有经过验证的愿景进行创业，可能是创业公司失败最主要的原因之一。这也是为什么精益创业运动提倡收集数据并基于经验进行决策。根据经验收集数据，根据数据做出决策，从而减少开发那些没人用的东西的风险。

提示

如果你是乔布斯或者巴菲特，追随你的直觉也许是一个不错的主意，可惜我们大部分人都不是。所以我们的决策应该基于数据，而不是直觉。

改变创业公司的决策模式也许会比较困难，而且也超越了本书的范围，不过我强烈建

议你阅读有关精益创业哲学的书。^[30,9]即使你不能改变企业现在的决策模式，你也应该收集一些数据，做一些客户访谈，搞一些 A/B 测试，试着帮助你的合作伙伴，确保自己不是在做无用功。

另外一个减少工作范围的方式是遵循二八法则，知道什么时候停下工作，而不是为了一点边际效益强制工作。二八法则是说 80% 的价值由 20% 的努力创造。神奇的是，二八法则在软件开发的很多地方都有体现。

- 80% 的功能花费 20% 的时间
- 80% 的代码需要 20% 的时间
- 80% 的用户只使用 20% 的功能；事实上，研究显示，很多系统有一半的功能从未使用。^[110]
- 80% 的文档价值在 20% 的内容上
- 80% 的 BUG 来自于 20% 的代码
- 80% 的代码变更集中在 20% 的代码上

虽然二八法则很简单，但是真正理解了它，可以避免花费时间去做那些锦上添花的事，让你在事情“刚刚完成”的时候就停止工作，而不会为了达到所谓的 100 分无休止地投入。二八法则是一种实用主义哲学。在很多地方都会用到二八法则，运用二八法则减少工作量的思路有：

- 和利益相关者一起讨论，将新特性的范围缩小至 80%，延迟最困难（代价最大）的部分功能的交付时间。在实现完整功能之前，尽早实现一些最基本的功能，以进行 A/B 测试，收集用户反馈。这种最小可用产品的方法适用于任何特性，有些时候，这个最小可用产品就是真实用户所需要的全部。
- 尽可能保持功能最小化及简单化。添加新特性的时候使用 A/B 测试，确保这些特性有用并能产生期望的价值。根本不会被调用的代码是某种形式的技术债，不会被使用的功能应该被删除以降低技术债务。记住，少即是多，特别对于一家创业公司而言。
- 确保只实现那些绝对必要的代码，不要添加那些“有也不错”的参数、类和方法。所有这些“然而并没有什么卵用”的代码都需要测试、文档、理解和管理。少即是多。

- 测试覆盖到 85%~90%的代码即可，而不是 100%的代码测试覆盖。有些代码块比其他地方更难以测试，测试这 10%的代码可能有些得不偿失。
- 创建文档的时候，关注主要信息及高层视图，不必为所有方面创建文档。多画架构图，一图真的胜千言。如果你觉得你的文档是完整的，那么你极可能浪费了 80%的时间。
- 如果不出故障就不要试图修复代码。需要修改代码的时候顺便重构这些代码。老代码如果不需要修改就不要动它。你为什么想要重构一个几个月都没人碰的类？就是为了感觉更好一点？我知道这句话听起来有点刺耳，不过这么做看起来真的有点强迫症，对创业公司而言是徒增负担。
- 将手头的任务区分为“我必须做的”和“我想要做的”，后者会产生大量额外的工作量。工程师们爱学习也爱开发——这样会产生一种偏见，就是只喜欢开发新的而不愿意复用旧的，喜欢尝试新技术而不是用自己熟悉的技术开发。结果就是，在工作中，我们总是追逐最新最酷的技术、框架及模式，而不是用最好的工具。此外，我们又足够聪明，总是能想到办法证明自己的选择对公司而言是最好的。看穿这种偏见可以帮助你减少大量的不必要的工作。
- 不到万不得已不要去伸缩或者优化。记住，即使你认为自己必须要做，也依然可能落入“我想要做”的陷阱。你不需要对你参与的每个项目都进行水平伸缩，大多数时候这都是浪费时间。据统计，90%拿到种子投资的创业公司都倒闭了。这个统计数据也适用于各种孵化器里的创业公司。剩下 10%的幸存者，绝大多数永远也不会伸缩到几十台机器的规模，不需要考虑水平伸缩。如果你在一家创业公司工作，你可以规划伸缩性，但是要尽可能地推迟实现而将精力集中于最紧急的需求上，比如确保你开发的产品是用户真实需要的。

工程师是一群充满激情乐观向上的人，这样的人做范围管理会格外困难。我们想要把所有的事情都搞定，想要所有的事情都做到完美，相信明天下午就能完成所有的事情。我们希望我们的 UI 是漂亮的，后端是优化的，数据是一致的，代码是干净的。不幸的是，除非你有无限的资源或者无穷的时间，否则你总要牺牲某些方面为更重要的事情让步。

影响成本

还有一种方式可以平衡工作量允许创业公司进行伸缩：增加成本从而减少工作量。你

可以将任务和责任委派给其他人或者第三方公司,从而减少你自己的工作范围。如果你的工作太多自己做不完,所有的工作又真的需要做,这些工作也没办法用自动化工具完成,截止时间也不能推后,那么你应该考虑将工作委派出去。

将任务委派给团队里的其他人,你就增加了你的部门的伸缩性。如果你是完成某项任务的唯一人选,那么你就会成为瓶颈并存在单点的风险。针对一项特定的任务让团队里的多个人都拥有处理的能力,这个工作就可以分配给多个人,而不会让自己或某个人成为瓶颈。另外,同一个任务让多个人参与,这样在这个领域获得突破和创新的概率会更大。例如,你的搭档可能会发现一个针对工作流程的自动化或者优化的方法,而你可能永远都不会想到。

为了能更容易地将任务和责任分配给团队的其他成员,你需要确保人们对不同的任务和应用的的不同部分都熟悉。因此,你需要让团队成员积极的彼此分享知识并更紧密的合作。这里,给出一些实践,有助于项目内部分享知识和责任。

- **结对编程**

两个工程师针对一个任务合作工作。虽然这种做法看起来有点低效,但是结对编程可以实现更高质量的设计、更少的 BUG,以及更紧密的合作。我并不是推荐所有时间都采用结对编程,但是每周有一天进行结对编程也许是一个分享知识、理解系统、互相学习的好办法。这也是指导团队新成员的一种非常好的方式,新成员可以直观地看到老员工如何思考问题,如何解决问题。

- **临时讨论会**

临时组织,用白板或者纸和笔,讨论问题,头脑风暴,获取更好的方案。

- **持续代码审查**

团队成员彼此交叉审查代码。代码审查不但可以提高代码质量,同时也是增强合作分享知识的好办法。彼此交叉代码审查让工程师之间彼此提供反馈,落实最佳实践,同时也可以促使工程师不断做出改变,持续进步。

增加成本降低工作负荷的另一种方式是购买第三方服务或者商业工具。通过第三方服务增加团队处理能力有一个非常好的例子,就是使用第三方的监控和报警工具。如果你想自己开发一个监控报警系统,你可能需要花费几个月的时间才能得到一个可用可伸缩的系统。但是,如果你决定部署一个满足同样需求的开源系统,你只需要数天的时间就可以让

系统运行起来，但是后续你还要花时间去维护它。如果你决定用第三方的工具，你就可以用花钱的方式节省开发维护这些系统的时间。通过使用第三方的监控服务，持续的金钱投入可以显著代替初始的时间成本及后续的维护时间。类似地，也可以使用各种成熟的商业工具、数据存储、缓存、工作流引擎、视频会议服务，从而降低工作负荷或者增加生产效率。

提示

工程师喜欢开发软件。这种喜欢开发新东西的特点让我们产生一种偏见，就是热衷开发胜过复用。开发监控服务、分析报警平台，甚至数据存储系统只是重复发明轮子的另一种形式。建议你在投入开发之前，先检查一下有没有已经存在的東西可以免费用，或者可以花钱买。

增加成本从而降低工作量的做法通常超出了工程师的权力范围，这些情况要汇报给相关的业务领导。完全不必做这类工作是改善伸缩性的一种手段，可以让你百分百地关注用户的需求。

影响计划

影响项目管理的三个杠杆的最后一个影响计划。和影响成本类似，决定某个功能何时发布通常不在工程师的掌控之内，但是你还是可以通过向业务领导提供某些反馈在某种程度上影响计划。大多数时候，截止日期和功能发布的命令都是可以谈判的，也会根据成本进行考量。某些特别的情况下，比如公司需要应对一个激烈的市场竞争，或者公司签了合同需要按时交付产品，那么你可能就要面对一个难以变更的截止日期，不过大多数情况下，事情总是可以商量的。

需要澄清一下，我不是说你该不遵守截止日期。延期发布通常都是糟糕的，对公司也会有伤害。我是说，在创业公司工作，你和决策者的距离会比较近，你可以在决策上影响交付的成果和时间。不是消极地听从指令，而是积极地向业务领导者提供反馈，让他们知道哪些功能是比较容易快速开发的，哪些功能是比较困难有风险的。这样，领导者就可以更好地理解成本，正确地评估任务的优先级。

为了持续提供反馈，我建议发布节奏更快速，每次发布的東西少一点，这样可以更快

速地获得用户反馈，决定是否还要继续原来的开发计划，是否有必要变更开发方向做点别的什么。快速学习是精益创业方法论的精髓。你频繁发布、收集反馈和数据，然后决定下一步的动向，而不是提前做好全部的计划。将产品开发分割成较小的迭代，可以减少风险降低出错的成本，风险和出错对于创业公司而言几乎是不可避免的。

比如，如果要扩展一个电商平台允许外部供应商开通他们自己的在线店铺，可以将所有的需求都列出来，制订计划，然后基于这个计划开发 3 到 6 个月，最后将新功能发布上线。开发的这几个月期间，没有任何用户反馈，感觉像悬在真空中开发。既没有办法知道用户是否喜欢你开发的这些功能，又没有办法知道用户想要的其他功能。一次发布需要的时间越长，开发出的东西不符合用户需求的风险越大。

更好的开发方式是将需求按照类似分镜头剧本那样的方式进行分割，将发布控制在一个尽可能小的范围内，每一次发布都基于上次发布的用户反馈进行开发。通过更加快速的发布，你有机会和用户进行频繁的互动，进行调查，收集 A/B 测试的数据，基于这些信息，可以重新定义待开发的功能列表，而不是按照事前的规划全部开发。将一个大的功能分割成小的功能，依然遵循二八定律，你会发现已经开发出来的功能对用户而言通常已经足够用了，待开发的各种功能其实并没有特别强烈的需求。

此外，将功能分割成小块后，你可以使用 mock 的方法，特别是在创业公司的早期。mock 是指并不会真正实现的功能，但是这个功能会呈现给用户，以此判断用户是否感兴趣进而决定是否要真正实现这个功能。

比如，你想实现一个人工智能算法，可以自动给卖家上传的商品图片打上标签，但是这个功能可能需要几个月甚至几年的时间才能开发完成。那么就可以用 mock 的方法试验一下，先在数据库里随机选择一些商品图片作为例子，然后让员工在没有人工智能算法的情况下对这些图片手工打标签，最后通过 A/B 测试的方法分析卖家的行为及对搜索引擎优化的影响。通过这种 mock 的方式进行数据收集，创业公司可以以更快的方式（只需几个星期）获取关于这个功能更多的有价值的信息，基于这些信息，你再决定是否继续开发这个功能，还是去做点别的。

根据公司的情况及你的职位，你可能很容易就控制范围、成本或者计划。通过各种权衡及向业务领导进行反馈，你可以更好地平衡工作负荷以避免过度工作。

伸缩敏捷团队

最后要强调的是伸缩敏捷团队的挑战。随着公司的发展，需要雇佣更多的工程师、项目经理、产品经理、系统管理员去持续开发产品。不幸的是，敏捷团队的伸缩要困难得多，你不能仅仅通过增加人手去伸缩团队。一个 8 人团队做的事交给一个几十人或者几百人的团队就没法做。

增加人手

首先需要注意的一个事实是，团队沟通成本的增加不是线性的。在一个 5 个人的团队里，你几乎总是可以立即知道发生了什么事；哪些东西正在被改变；有什么事情需要发起一次讨论；选择了哪种设计、标准及实践。随着团队人员不断增加，沟通的路径也会迅速增加，不可能让每个人都能及时了解最新的状况。图 9-8 所示为不同团队规模下沟通路径的情况。

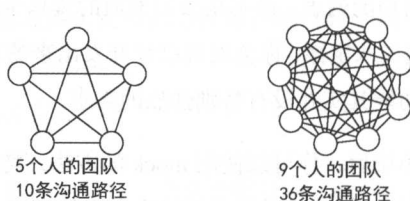


图 9-8 沟通路径数量

解决这个问题的常用方式是将大团队拆分成一些小团队，每个团队的人数为 4~9 人，每个团队的功能职责也不同。一个测试小组，一个系统管理小组，几个专注不同领域的开发小组，如果需要还可以设立项目管理小组及产品经理小组。这种方式看起来是一种可持续伸缩的好办法，但却不可避免地会出现在不同团队之间踢皮球、政治斗争、互相指责、推卸责任等一系列现象。

这种导致工作效率低下的情况是因为团队是根据工作职能划分的，而不是基于产品开发划分的。后果就是，产品开发生命周期被切分成一段一段，每个团队都参与其中的一个阶段，每个团队都只关注自己涉及的那一部分，而不会考虑整体的产品目标。这类组织架构中，产品经理将需求交给具体的工程师，工程师写代码，然后交给测试及运维工程师。等出了问题的时候，就开始互相指责，于是不得不制定更严格的流程制度，开发变得更困

难，所有参与其中的人都只想着如何尽快脱身。

幸运的是，可伸缩的软件工程团队还有一种更好的方式：创建跨职能的团队。^[11]在这种团队里，不会只包含测试工程师或者开发工程师，而是按照产品或者服务组建团队。例如，某个特定的团队负责维护 Checkout 功能，就会包含一个设计师，一个 Web 工程师，一个系统管理员，三个后端工程师，几个前端工程师。这个团队可以自主完成设计、开发、部署等一系列工作。理想情况下，这个团队也能自己收集反馈，分析数据，领导产品开发，无须等待任何人的指令。

减少团队之间的依赖，给他们更多的自主性，开发过程会变得更加独立。从这方面考虑，组织层面的伸缩性和应用的伸缩性其实很类似。你需要增加更多员工（服务器）并在他们之间分配工作。同样，要想最大化组织吞吐能力，员工也需要能够不必等待其他员工，自主做出决定。另外，他们需要获得技能、工具，以及授权（代码和数据），以便彼此之间的沟通尽量少。

这种创建跨职能团队对技术部门进行伸缩的方式比较适合创业公司，特别是那些使用面向服务（微服务）架构的公司，系统被设计成一组松耦合的 UI 与 Web 服务，这些组件可以被不同的团队独立开发和部署。最好是让团队负责一个端对端的产品，比如 Checkout，包括了 UI、前端、后端、数据存储，后面随着应用的不断发展，也许需要创建一个专门的团队负责 Checkout 的 UI 及其他 Checkout 服务管理。通过将这个团队拆分成两个，你可以投入更多人手到 Checkout 这个产品上，但是同时，你也制造了跨团队的依赖，当 UI 同学需要等待服务变更的时候，开发进度就慢下来。

流程与创新

伸缩技术部门的另一个重要工作是正确平衡过程、标准，以及团队自主性的关系。随着组织的不断成长，你也许想开发一些流程确保组织行为一致，每个人都遵循最佳实践。例如，你可能需要所有的团队都有一个通讯录，确保 24 小时随叫随到处理产品故障。你可能想要每个服务都有一个业务持续计划，如果系统故障或者崩溃能够快速恢复，你可能也想制定一套代码标准、文档标准、分支策略、敏捷过程、审查过程、自动化需求、结构模型指南、审计跟踪需求、测试最佳实践，等等。

有一件重要的事情需要注意，当你制定了标准和流程并从中获益的时候，同时你也牺

牲了团队的自主性、弹性，以及创新的精神。

举个例子，你制定了一个规范，为了保证业务持续可用，强制让每个产品都有数据复制和数据热备切换功能。这个规范也许对保证系统 99.999% 可用很有帮助，但是同时可能导致团队产出急剧下降。本来花几个星期就能开发出一个非核心服务，但是为了保证业务的持续可用，可能要花上几个月的时间。在某些情况下，这可能是件好事，但在不确定需求是否真实的情况下，都强制执行这一规范，必然会使得尝试和学习的代价变得很高。

流程和标准是公司成长的重要组成部分，但是你必须要使它们保持精简和富有弹性，不要让这些流程标准对公司的生产率、敏捷性和企业文化产生负作用。

团结的文化

另一个技术部门伸缩性的重要方面是将技术团队团结到一组共同的目标上并构建一种良好的工程技术文化。没有团结，各个团队都自行其是，会将产品引导到不同的方向，关注的优先级各不相同，彼此之间不停地发生冲突。作为一个工程师，一个领导，一个业务决策者，将团队团结起来，才能让创业公司获得更好的前进动力及更高效的产出。

图 9-9 所示为技术部门团结一致的重要性。如果各个团队是不团结的，他们就会走向不同的方向。结果就是整体的方向不明确，整体的步调不统一。相对应的，如果所有人都团结一致往一个方向努力，就不会彼此互相抵消努力，前进的动力更加强劲。

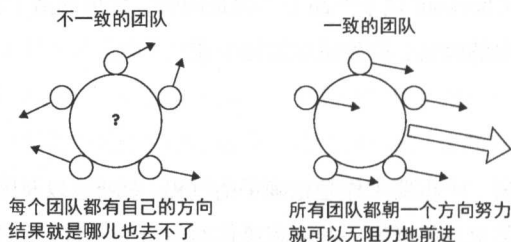


图 9-9 一致的影响

要想让团队之间团结一致，需要建设一种谦逊、尊重、诚信的组织文化，每个工程师都笃信“我们团结一致共同努力”。作为一个好的团队成员，总是相信公司利益优于团队利益，团队利益优于个人利益。

无论你是 CTO、经理、还是工程师，在你沟通需求的时候，在你寻求妥协的时候，在你理解别人观点的时候，总会以各种各样的方式受公司文化的影响并影响公司的文化。一个好的工程师文化是建立在互相尊重、信任、关注的基础上的。^[11]没有好的工程师文化，就会陷入一种恶性循环中：差的工程师导致差的文化，差的文化招来更差的工程师，公司陷入政治斗争的泥潭，在彼此扯皮拖后腿中消耗了无穷的精力。

为了让员工高效、健康、快乐，你需要创建这样一种工作环境：每个人都感到安全并被接纳，彼此之间乐于支持和帮助。好消息是，每个人都可以培育和强化好的企业文化，而不必非得 CEO 才能做到。

小结

这些年的工作让我认识到，伸缩性是一个深入而艰难的课题。本书中涉及很多伸缩性的方面，虽然仍然只是触及一些皮毛，不过相信可以帮助你建立伸缩性的一个整体概念，引导你学习更深入的知识。

如果你对从组织视角考虑伸缩性很有兴趣，我强烈建议你学习更多“精益创业”的思想，^[30,9]建设一种好的工程师文化^[11]及自动化过程。^[4]每个话题都深似海，靠这小小一章根本不可能覆盖全部。

从个人层面，我的建议是实事求是，永远不要停止学习。开发软件，特别是可伸缩的软件其实是一个权衡的游戏，没有所谓的正确与否。不断学习，然后自己决定哪些规则应该遵守，哪些规则可以打破。

A

推荐阅读

下面列出的推荐阅读清单可以帮助读者更好地搭建构建可伸缩的 Web 应用的知识体系。

本书并没有直接引用以下内容，仅附在全书最后，作为推荐阅读。

Books

1. Robert Cecil Martin (2002) *Agile Software Development, Principles, Patterns, and Practices*
2. Eric Evans (2003) *Domain-Driven Design: Tackling Complexity in the Heart of Software*
3. Dean Leffingwell, Don Widrig (2010) *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*
4. John Allspaw, Jesse Robbins (2010) *Web Operations: Keeping the Data on Time*
5. Diomidis Spinellis, Georgios Gousios (2009) *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*
6. John Allspaw (2008) *The Art of Capacity Planning: Scaling Web Resources*
7. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*
8. Steve Souders (2009) *Even Faster Web Sites: Performance Best Practices for Web Developers*
9. Curt Hibbs, Steve Jewett, Mike Sullivan (2009) *The Art of Lean Software Development*
10. Martin Fowler (2002) *Patterns of Enterprise Application Architecture*
11. Brian Fitzpatrick, Ben Collins-Sussman (2012) *Team Geek*
12. Alvaro Videla, Jason Williams (2012) *RabbitMQ in Action: Distributed Messaging for Everyone*
13. Ian Molyneaux (2009) *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*
14. Gary Mak (2008) *Spring Recipes: A Problem-Solution Approach*
15. Chad Fowler (2009) *The Passionate Programmer: Creating a Remarkable Career in Software Development*

16. Jeremy Zawodny, Derek Balling (2004) High Performance MySQL: Optimization, Backups, Replication, Load Balancing & More
17. Ivo Jansch (2008) *PHP Architect's Guide to Enterprise PHP Development*
18. Paul Allen, Joseph Bambara (2007) Sun Certified Enterprise Architect for Java EE Study Guide
19. Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein (2003) Introduction to Algorithms
20. David Chappell, Tyler Jewell (2002) Java Web Services
21. Vivek Chopra, Sing Li, Jeff Genender (2007) *Professional Apache Tomcat 6*
22. Glen Smith, Peter Ledbrook (2009) Grails in Action
23. Chuck Lam (2010) Hadoop in Action
24. Opher Etzion, Peter Niblett (2011) Event Processing in Action
25. Bruce Snyder, Dejan Bosanac, Rob Davies (2011) ActiveMQ in Action
26. Henry Liu (2009) *Software Performance and Scalability: A Quantitative Approach*
27. Bill Wilder (2012) Cloud Architecture Patterns
28. John Arundel (2013) Puppet 3 Beginner's Guide
29. Jeffrey Barr (2010) Host Your Web Site in the Cloud: Amazon Web Services Made Easy
30. Eric Ries (2011) The Lean Startup
31. Arnon Rotem-Gal-Oz (2012) SOA Patterns
32. Gregor Hohpe, Bobby Woolf (2003) Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions
33. David S. Linthicum (2009) Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide
34. Bashar Abdul-Jawad (2008) Groovy and Grails Recipes
35. Charles Bell, Mats Kindahl, Lars Thalmann (2010) MySQL High Availability
36. Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra (2004) Head First Design Patterns
37. Robert C. Martin (2008) Clean Code: A Handbook of Agile Software Craftsmanship
38. Steve McConnell (2004) *Code Complete, Second Edition*

39. Dominic Betts, Julián Domínguez, Grigori Melnik, Fernando Simonazzi, Mani Subramanian, Microsoft (2012) Exploring CQRS and Event Sourcing A Journey into High Scalability, Availability, and Maintainability with Windows Azure; <http://msdn.microsoft.com/en-us/library/jj554200.aspx>
40. Frederick Brooks (1995) The Mythical Man-Month: Essays on Software Engineering
41. Martin L. Abbott, Michael T. Fisher (2009) The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise
42. Duane Wessels (2001) Web Caching
43. Martin Abbott, Michael Fisher (2011) Scalability Rules: 50 Principles for Scaling Web Sites
44. Kyle Banker (2011) MongoDB in Action
45. Joel Spolsky (2004) Joel on Software
46. Jim Webber, Savas Parastatidis, Ian Robinson (2010) REST in Practice
47. Flavio Junqueira, Benjamin Reed (2013) ZooKeeper
48. Peter Membrey, David Hows, Eelco Plugge (2012) Practical Load Balancing: Ride the Performance Tiger
49. Steve Souders (2007) High Performance Web Sites
50. Josiah L. Carlson (2013) Redis in Action
51. Martin Kalin (2013) Java Web Services: Up and Running, 2nd Edition

白书皮

- w1. Jeffrey Dean, Sanjay Ghemawat (2004) MapReduce: Simplified Data Processing on Large Clusters
http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf
- w2. Floris Engelbertink, Harald Vogt (2010) How to Save on Software Maintenance Costs
http://www.omnext.net/downloads/Whitepaper_Omnext.pdf
- w3. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, Member (2006) CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code
<http://pages.cs.wisc.edu/~shanlu/paper/TSE-CPMiner.pdf>

- w4. Angela Lozano, Michel Wermelinger (2010) Tracking Clones' Imprint
<http://released.info.ucl.ac.be/pmwiki/uploads/Publications/TrackingClonesImprint/clonesImprint.pdf>
- w5. NASA Office of Chief Engineer (2009) NASA Study on Flight Software Complexity
http://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf
- w6. Jakub Łopuszanski (2013) Algorithm for Invalidation of Cached Results of Queries to a Single Table
http://vanisoft.pl/~lopuszanski/public/cache_invalidation.pdf
- w7. Google, Inc. (2012) Spanner: Google's Globally-Distributed Database
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/spanner-osdi2012.pdf
- w8. *Solarflare* (2012) *Cisco and Solarflare Achieve Dramatic Latency Reduction for Interactive Web Applications with Couchbase, a NoSQL Database*
http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-708169.pdf
- w9. *Couchbase* (2013) *Dealing with Memcached Challenges*
http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Couchbase_Whitepaper_Dealing_with_Memcached_Challenges.pdf
- w10. Gregor Hohpe (2006) Programming Without a Call Stack: Event-Driven Architectures
<http://www.eaipatterns.com/docs/EDA.pdf>
- w11. Matt Welsh (2000) The Staged Event-Driven Architecture for Highly-Concurrent Server Applications
<http://www.eecs.harvard.edu/~mdw/papers/quals-seda.pdf>
- w12. Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, Antony Rowstron (2013) *Nobody Ever Got Fired for Buying a Cluster*
<http://research.microsoft.com/pubs/179615/msrtr-2013-2.pdf>
- w13. Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, Antony Rowstron (2009) *Migrating Server Storage to SSDs: Analysis of Tradeoffs*
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.2362&rep=rep1&type=pdf>
- w14. Jiri Simsa, Randy Bryant, Garth Gibson, Jason Hickey (2013) Scalable Dynamic Partial Order Reduction
<http://www.pdl.cmu.edu/PDL-FTP/Storage/scalablePOR.pdf>

- w15. Ariel Rabkin, Randy Katz (2012) How Hadoop Clusters Break
<http://www.cs.princeton.edu/~asrabkin/papers/software12.pdf>
- w16. Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, Ion Stoica (2012) *Probabilistically Bounded Staleness for Practical Partial Quorums*
<http://arxiv.org/pdf/1204.6082.pdf>
- w17. Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden (2013) MDCC: Multi-Data Center Consistency
<http://arxiv.org/pdf/1203.6049.pdf>
- w18. Google, Inc. (2011) Megastore: Providing Scalable, Highly Available Storage for Interactive Services
<http://pdos.csail.mit.edu/6.824-2012/papers/jbaker-megastore.pdf>
- w19. Brenda M. Michelson (2006) *Event-Driven Architecture Overview*
<http://www.omg.org/soa/Uploaded%20Docs/EDA/bda2-2-06cc.pdf>
- w20. Facebook Data Infrastructure Team (2010) Hive: A Petabyte Scale Data Warehouse Using Hadoop
http://people.cs.kuleuven.be/~bettina.berendt/teaching/2010-11-2ndsemester/ctdb/petabyte_facebook.pdf
- w21. Ian Foster, Yong Zhao, Ioan Raicu, Shiyong Lu (2008) *Cloud Computing and Grid Computing 360-Degree Compared*
<http://arxiv.org/ftp/arxiv/papers/0901/0901.0131.pdf>
- w22. Daniel J. Abadi (2012) *Consistency Tradeoffs in Modern Distributed Database System Design*
<http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf>
- w23. Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica (2013) HAT, not CAP: Highly Available Transactions
<http://arxiv.org/pdf/1302.0309.pdf>
- w24. Stephan Muller (2012) The CAP-Theorem & Yahoo's PNUTS
<http://www.math.hu-berlin.de/~muellste/CAP-PNUTS-Text.pdf>
- w25. Eric Brewer (2012) CAP Twelve Years Later: How the "Rules" Have Changed
http://www.realtechsupport.org/UB/NP/Numeracy_CAP+12Years_2012.pdf
- w26. Microsoft Research (2012) *Cloud Types for Eventual Consistency*
<http://research.microsoft.com/pubs/163842/final-with-color.pdf>
- w27. Avinash Lakshman, Prashant Malik (2009) Cassandra: A Decentralized Structured Storage System
<http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf>

- w28. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber (2006) Bigtable: A Distributed Storage System for Structured Data
http://static.usenix.org/events/osdi06/tech/chang/chang_html/?em_x=22
- w29. Ryan Thompson, T.C. Friel (2013) The Sustainability of Cloud Storage
<http://sais.aisnet.org/2013/ThompsonFriel.pdf>
- w30. Edward P. Holden, Jai W. Kang (2011) *Databases in the Cloud: A Status Report*
<http://sigite2011.sigite.org/wp-content/uploads/2011/10/session11-paper02.pdf>
- w31. Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, Alexander Rasin (2009) *HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads*
http://www-master.ufr-info-p6.jussieu.fr/2009/Ext/naacke/grbd2010/extra/exposes2010/C3_VLDB09_HadoopDB.pdf
- w32. Brian Holcomb (2013) *NoSQL Database in the Cloud: Riak on AWS*
http://media.amazonwebservices.com/AWS_NoSQL_Riak.pdf
- w33. Miles Ward (2013) *NoSQL Database in the Cloud: MongoDB on AWS*
http://media.amazonwebservices.com/AWS_NoSQL_MongoDB.pdf
- w34. Matt Tavis, Philip Fitzsimons (2012) *Web Application Hosting in the AWS Cloud Best Practices*
http://media.amazonwebservices.com/AWS_Web_Hosting_Best_Practices.pdf
- w35. Jinesh Varia (2011) *Architecting for the Cloud: Best Practices*
http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
- w36. Jeff Barr, Attila Narin, and Jinesh Varia (2011) *Building Fault-Tolerant Applications on AWS*
http://media.amazonwebservices.com/AWS_Building_Fault_Tolerant_Applications.pdf
- w37. *Amazon Web Services (2010) AWS Security Best Practices*
http://media.amazonwebservices.com/Whitepaper_Security_Best_Practices_2010.pdf
- w38. Jinesh Varia (2008) *Cloud Architectures*
http://media.amazonwebservices.com/AWS_Cloud_Architectures.pdf
- w39. Amazon.com (2007) *Dynamo: Amazon's Highly Available Key-Value Store*
<http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf>

- w40. Edward Curry, Desmond Chambers, Gerard Lyons (2004) *Extending Message-Oriented Middleware Using Interception*
http://www.edwardcurry.org/web_publications/curry_DEBS_04.pdf
- w41. Sharma Chakravarthy, Raman Adaikkalavan (2007) *Ubiquitous Nature of Event-Driven Approaches: A Retrospective View*
<http://drops.dagstuhl.de/opus/volltexte/2007/1150/pdf/07191.ChakravarthySharma.Paper.1150.pdf>
- w42. Daniel Ford, Francois Labelle, Florentina Popovici, Google (2010) *Availability in Globally Distributed Storage Systems*
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//pubs/archive/36737.pdf
- w43. Daniel Peng, Frank Dabek, Google (2010) *Large-Scale Incremental Processing Using Distributed Transactions and Notifications*
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//pubs/archive/36726.pdf
- w44. Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, Google (2003) *The Google File System*
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/gfs-sosp2003.pdf
- w45. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, Google (2010) *Pregel: A System for Large-Scale Graph Processing*
http://kowshik.github.io/JPregel/pregel_paper.pdf
- w46. Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Google (2010) *Dremel: Interactive Analysis of Web-Scale Datasets*
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//pubs/archive/36632.pdf
- w47. Sergey Brin, Lawrence Page, Google (1998) *The Anatomy of a Large-Scale Hypertextual Web Search Engine*
<http://ilpubs.stanford.edu:8090/361/1/1998-8.pdf>
- w48. Luiz André Barroso, Jeffrey Dean, Urs Hölzle, Google (2003) *Websearch for a Planet: The Google Cluster Architecture*
<http://www.eecs.harvard.edu/~dbrooks/cs246-fall2004/google.pdf>
- w49. Mike Burrows, Google, Inc. (2006) *The Chubby Lock Service for Loosely-Coupled Distributed Systems*
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//archive/chubby-osdi06.pdf

- w50. Raymond Cheng, Will Scott, Paul Ellenbogen, Arvind Krishnamurthy, Thomas Anderson (2013) *Radiatus: Strong User Isolation for Scalable Web Applications*
<http://www.cs.washington.edu/education/grad/UW-CSE-13-11-01.PDF>
- w51. Yahoo! Research (2010) *Feeding Frenzy: Selectively Materializing Users' Event Feeds*.
- w52. LinkedIn (2011) *Kafka: A Distributed Messaging System for Log Processing*
<http://research.microsoft.com/en-us/um/people/srikanth/netdb11/netdb11papers/netdb11-final12.pdf>
- w53. LinkedIn (2013) *Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph*
<http://0b4af6cdc2f0c5998459-c0245c5c937c5dedcca3f1764ecc9b2f.r43.cf2.rackcdn.com/11567-hotcloud13-wang.pdf>
- w54. AMQP.org (2011) *AMQP v1.0 Specification*
<http://www.amqp.org/sites/amqp.org/files/amqp.pdf>
- w55. DataStax (2014) *Apache Cassandra 2.0 Documentation*
<http://www.datastax.com/documentation/cassandra/2.0/pdf/cassandra20.pdf>
- w56. DataStax (2014) *CQL for Cassandra 2.0 Documentation*
<http://www.datastax.com/documentation/cql/3.1/pdf/cql31.pdf>
- w57. George Candea, Armando Fox from Stanford University (2003) *Crash-Only Software*
https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf
- w58. Konstantin V. Shvachko (2010) *HDFS Scalability: The Limits to Growth*
<https://www.usenix.org/legacy/publications/login/2010-04/openpdfs/shvachko.pdf>
- w59. Google, Inc. (2013) *MillWheel: Fault-Tolerant Stream Processing at Internet Scale*
<http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/41378.pdf>
- w60. Google, Inc. (2010) *FlumeJava: Easy, Efficient Data-Parallel Pipelines*
<http://pages.cs.wisc.edu/~akella/CS838/F12/838-CloudPapers/FlumeJava.pdf>
- w61. Ranjit Noronha, Dhabaleswar K. Panda (2008) *IMCa: A High Performance Caching Front-End for GlusterFS on InfiniBand*
<http://nowlab.cse.ohio-state.edu/publications/conf-papers/2008/noronha-icpp08.pdf>

- w62. Facebook (2013) Scaling Memcache at Facebook
https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf&sa=U&ei=gWJjU97pOeqxsQSDkYDAAG&ved=0CBsQFjAA&usg=AFQjCNGMeuWne9ywncbgux_XiZW6lQWHNw
- w63. Intel (2012) *Enhancing the Scalability of Memcached*
https://software.intel.com/sites/default/files/m/0/b/6/1/d/45675-memcached_05172012.pdf
- w64. Yahoo! (2010) *ZooKeeper: Wait-Free Coordination for Internet-Scale Systems*
https://www.usenix.org/legacy/event/usenix10/tech/full_papers/Hunt.pdf
- w65. University of Washington (2011) *Scalable Consistency in Scatter*
<http://homes.cs.washington.edu/~arvind/papers/scatter.pdf>
- w66. James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, Alistair Veitch (2012) *LazyBase: Trading Freshness for Performance in a Scalable Database*
<http://www.pdl.cmu.edu/PDL-FTP/Database/euro166-cipar.pdf>
- w67. Hyeontaek Lim, Bin Fan, David G. Andersen, Michael Kaminsky (2011) *SILT: A Memory-Efficient, High-Performance Key-Value Store*
<https://www.cs.cmu.edu/~dga/papers/silt-sosp2011.pdf>
- w68. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, David G. Andersen (2011) *Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS*
<http://sns.cs.princeton.edu/docs/cops-sosp11.pdf>
- w69. Microsoft (2007) *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*
<http://www.cs.cmu.edu/~15712/papers/isard07.pdf>
- w70. Facebook (2013) *TAO: Facebook's Distributed Data Store for the Social Graph*
<https://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/11730-atc13-bronson.pdf>
- w71. Facebook (2013) *Unicorn: A System for Searching the Social Graph*
<http://db.disi.unitn.eu/pages/VLDBProgram/pdf/industry/p871-curtiss.pdf>
- w72. Google (2014) *Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing*
<http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/42851.pdf>
- w73. The Ohio State University (2013) *Understanding the Robustness of SSDs under Power Fault*
<https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf>

- w74. Philippe Flajolet, Éric Fusy, Olivier Gandouet, Frédéric Meunier (2007) *HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm*
<http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>
- w75. Google, Inc. (2013) *HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm*
<http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf>
- w76. Martin Fowler (2004) *Inversion of Control Containers and the Dependency Injection Pattern*
<https://blog.itu.dk/MMAD-F2013/files/2013/02/3-inversion-of-control-containers-and-the-dependency-injection-pattern.pdf>
- w77. Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, Andrew Stewart (2011) *Disruptor: High Performance Alternative to Bounded Queues for Exchanging Data Between Concurrent Threads*
<http://disruptor.googlecode.com/files/Disruptor-1.0.pdf>
- w78. Pat Helland (2009) *Building on Quicksand*
http://blogs.msdn.com/cfs-file.ashx/__key/communityserver-components-postattachments/00-09-20-52-14/BuildingOnQuicksand_2D00_V3_2D00_081212h_2D00_pdf.pdf
- w79. Mark Slee, Aditya Agarwal, Marc Kwiatkowski, Facebook (2007) *Thrift: Scalable Cross-Language Services Implementation*
<http://thrift.apache.org/static/files/thrift-20070401.pdf>

演讲

- t1. Robert C. Martin (2011) *Keynote Speech of Ruby Midwest: Architecture the Lost Years*
- t2. Renat Khasanshyn (2012) *CouchConf San Francisco*
<http://www.couchbase.com/presentations/benchmarking-couchbase>
- t3. Google, Inc. (2012) *F1: The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business*
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//pubs/archive/38125.pdf
- t4. Andy Parsons (2013) *Lessons on Scaling Rapidly-Growing Startups in the Cloud*
<http://java.dzone.com/articles/lessons-scaling-rapidly>

- t5. Sean Cribbs (2012) *Fear No More: Embrace Eventual Consistency*
http://qconsf.com/sf2012/dl/qcon-sanfran-2012/slides/SeanCribbs_FearNoMoreEmbraceEventualConsistency.pdf
- t6. Robert Hodges (2013) *State of the Art for MySQL Multi-Master Replication*
http://www.percona.com/live/mysql-conference-2013/sites/default/files/slides/mysql-multi-master-state-of-art-2013-04-24_0.pdf
- t7. Jay Patel (2013)
<http://www.slideshare.net/jaykumarpatel/cassandra-data-modeling-best-practices>

链接

- L1. Windows Azure Queues and Windows Azure Service Bus Queues: Compared and Contrasted
<http://msdn.microsoft.com/en-us/library/windowsazure/hh767287.aspx>
- L2. Thomas Bayer (2013) Broker Wars
<http://www.predic8.com/activemq-hornetq-rabbitmq-apollo-qpuid-comparison.htm>
- L3. Amazon SQS Documentation
<http://aws.amazon.com/documentation/sqs/>
- L4. Google Trends of ActiveMQ and RabbitMQ Searches
<http://www.google.com/trends/explore?q=activemq%2C+rabbitmq%2C+zeromq%2C+hornetq#q=activemq%2C%20rabbitmq&date=7%2F2008%2061m&cmpt=q>
- L5. Datadog
<https://www.datadoghq.com/product/>
- L6. Martin Fowler (2011) The LMAX Architecture
<http://martinfowler.com/articles/lmax.html>
- L7. Martin Fowler (2005) Event Sourcing
<http://martinfowler.com/eaaDev/EventSourcing.html>
- L8. Mitchell Anicas (2014) How to Use Logstash and Kibana to Centralize Logs on Ubuntu 14.04
<https://www.digitalocean.com/community/tutorials/how-to-use-logstash-and-kibana-to-centralize-and-visualize-logs-on-ubuntu-14-04>

- L9. Logstash (2013) Introduction
<http://logstash.net/docs/1.4.2/tutorials/getting-started-with-logstash>
- L10. Luu Duong's Blog (2009) Applying the "80-20 Rule" with The Standish Group's Statistics on Software Usage
<http://luuduong.com/blog/archive/2009/03/04/applying-the-quot8020-rulequot-with-the-standish-groups-software-usage.aspx>
- L11. Spotify (2014) Spotify Engineering Culture
<https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/>
- L12. *The Netflix Tech Blog* (2010) *Chaos Monkey Released into the Wild*
<http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>
- L13. Azure SQL Database Elastic Scale Overview
<http://azure.microsoft.com/en-us/documentation/articles/sql-database-elastic-scale-introduction/>
- L14. *The Netflix Tech Blog* (2013) *Astyanax Update*
<http://techblog.netflix.com/2013/12/astyanax-update.html>
- L15. Red Hat Storage Server NAS Takes on Lustre, NetApp
http://www.theregister.co.uk/2012/06/27/redhat_storage_server_2_launch/
- L16. Zookeeper
<http://zookeeper.apache.org/>
- L17. Curator
<http://curator.apache.org/>
- L18. Amazon API (2012) Elastic Load Balancer LB Cookie Stickiness
http://docs.aws.amazon.com/ElasticLoadBalancing/latest/APIReference/API_CreateLBCookieStickinessPolicy.html
- L19. F5 DevCentral (2013) Back to Basics: The Many Faces of Load Balancing Persistence
<https://devcentral.f5.com/articles/back-to-basics-the-many-faces-of-load-balancing-persistence>
- L20. Amazon (2013) Creating Latency Resource Record Sets
<http://docs.aws.amazon.com/Route53/latest/DeveloperGuide/CreatingLatencyRRSets.html>
- L21. Amazon (2012) Multi-Region Latency Based Routing Now Available for AWS
<http://aws.amazon.com/blogs/aws/latency-based-multi-region-routing-now-available-for-aws/>

- L22. Amazon (2014) Two New Edge Locations for CloudFront and Route 53
<http://aws.amazon.com/blogs/aws/two-new-edge-locations-for-cloudfront-and-route-53-taipei-and-rio-de-janeiro/>
- L23. Wikipedia, List of Managed DNS Providers
http://en.wikipedia.org/wiki/List_of_managed_DNS_providers
- L24. Cloudharmony blog (2012) Comparison and Analysis of Managed DNS Providers
<http://blog.cloudharmony.com/2012/08/comparison-and-analysis-of-managed-dns.html>
- L25. Citrix (2013) Citrix NetScaler
http://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/netscaler-data-sheet.pdf
- L26. F5 Networks (2013) Comparative Performance Report
<http://www.f5.com/pdf/reports/F5-comparative-performance-report-ADC-2013.pdf>
- L27. statisticshowto.com, Misleading Graphs: Real Life Examples
<http://www.statisticshowto.com/misleading-graphs/>
- L28. Gernot Heiser (2010) Systems Benchmarking Crimes
<http://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html>
- L29. Amazon, *Auto-scaling Documentation*
<http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/USBasicSetup-Console.html>
- L30. Amazon, *Auto-scaling Documentation*
<http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/as-register-lbs-with-asg.html>
- L31. highscalability.com (2013) Scaling Pinterest: From 0 to 10s of Billions of Page Views a Month in Two Years
<http://highscalability.com/blog/2013/4/15/scaling-pinterest-from-0-to-10s-of-billions-of-page-views-a.html>
- L32. highscalability.com (2010) 7 Lessons Learned While Building Reddit to 270 Million Page Views a Month
<http://highscalability.com/blog/2010/5/17/7-lessons-learned-while-building-reddit-to-270-million-page.html>
- L33. highscalability.com (2012) Tumblr Architecture: 15 Billion Page Views a Month and Harder to Scale Than Twitter
<http://highscalability.com/blog/2012/2/13/tumblr-architecture-15-billion-page-views-a-month-and-harder.html>

- L34. Charles Bretana via stackoverflow.com (2009) Does Anyone Have a Good Analogy for Dependency Injection?
<http://stackoverflow.com/questions/424457/does-anyone-have-a-good-analogy-for-dependency-injection>
- L35. highscalability.com (2010) Facebook at 13 Million Queries Per Second Recommends: Minimize Request Variance
<http://highscalability.com/blog/2010/11/4/facebook-at-13-million-queries-per-second-recommends-minimiz.html>
- L36. www.percona.com (2014) MySQL Ring Replication: Why It Is a Bad Option
<http://www.percona.com/blog/2014/10/07/mysql-ring-replication-why-it-is-a-bad-option/>
- L37. Pramod Sadalage, Martin Fowler (2012) Introduction to Polyglot Persistence: Using Different Data Storage Technologies for Varying Data Storage Needs
<http://www.informit.com/articles/article.aspx?p=1930511>
- L38. Fangjin Yang (2012) *Fast, Cheap, and 98% Right: Cardinality Estimation for Big Data*
<http://druidd.io/blog/2012/05/04/fast-cheap-and-98-right-cardinality-estimation-for-big-data.html>
- L39. *Stripe, API Libraries*
<https://stripe.com/docs/libraries>

任何大型网站都不是一蹴而就的，一定需要经过多年的技术磨练和沉淀。本书针对Web应用程序每一层的可伸缩性展开了全面细致的讨论，对互联网创业者的系统架构设计有莫大的帮助。

UCloud CEO 季昕华

互联网创业热，技术人才不足的问题凸显出来，其中最稀缺的，是能够构建可伸缩系统的工程师，在机会来临时不掉链子。但是，大部分程序员在职业生涯中并没有太多机会有这样的实战经验，这方面的技术文档也比较稀缺、零散。本书弥补了这一空白，面向创业公司技术人员，系统、全面且有针对性地总结可伸缩方面的原则和实践，兼顾开发、运维和团队等主题，有很多接地气的建议，语言风格明快，因此原著获得了很好的口碑。更为难得的是，译者本身也是这一领域的专家，在业界有“教授”的美誉，有力保证了本书成为一部经得起考验的佳作。

美团点评技术学院院长 刘江

2008年认识李智慧，从那时起，智慧参与阿里巴巴互联网基础技术平台的建设，并成为关键产品的代码贡献者。2014年智慧送了我一本他写的《大型网站技术架构：核心原理与案例分析》。翻看后，感觉智慧经历了这几年在大型网站的实践后，多了一份坚持总结和分享的能力。这是很多当时参与基础技术平台建设的同事所不具备的特质。近期，智慧告诉我他和另一位优秀的架构师何坤，一起翻译了本书。我翻看后，很佩服他俩在工作之余还能坚持翻译此书。书中内容涉及前端、服务、数据库、缓存、异步消息和搜索，非常全面。全世界的大型网站并不多，能参与其中建设的只是小部分工程师，希望他们解决问题的思路能给各位读者一些启发。

平安好医生CTO 王齐

互联网创业从0开始，之后迈出的每一步都是困难的。从0到1的过程不是瞬间完成，而是从0.01到0.02，直到1的蜕变。公司在不断发展与迭代的过程中，各种各样的技术问题也随之“野蛮”生长。如何解决系统的稳定性、可伸缩性等技术难题？本书或许能给你想要的答案。李智慧老师作为极客邦科技/InfoQ中国的专家讲师，长期致力于技术社区的发展。我愿推荐此书，和技术人共同成长。

极客邦科技/InfoQ中国创始人兼CEO 霍泰稳

构建一个“能用的”应用系统与构建一个能够随着业务发展而扩展的系统完全是两回事。要想在用户量和业务量快速增长的环境下保持系统的扩展性，需要工程师熟悉互联网业务中“扩展”与“伸缩”的常用招数，并能够根据情况选择最合适的方案。本书系统地描述了互联网应用中的扩展性，在系统设计原则、前后端设计、存储设计方面进行了全面的描述，并详尽讨论了“异步”与“缓存”这两个扩展性法宝的各种使用场景。希望在本书的帮助下，工程师能够快速掌握解决伸缩性问题的钥匙，早日成为可信赖的技术达人。

宜信宜人贷首席技术官 段念

和智慧在阿里共事多年，一起经历了阿里巴巴技术平台化战略的关键时期。互联网技术与传统软件技术相比，最大的不同可能就是如何处理高并发、大数据的挑战，而主要应对措施就是可伸缩的架构技术，期待本书成为互联网创业公司工程师的手边书，通过本书深入理解可伸缩系统架构的原理与设计。

同盾创始人 蒋韬

McGraw-Hill
全球智慧中文化

http://www.mheducation.com



策划编辑：刘 皎 @皎丫子
责任编辑：郑柳洁

欢迎投稿
Ljiao@phei.com.cn
010-88254395



上架建议：互联网创业>网站架构

ISBN 978-7-121-30112-4



定价：89.00元